OOCFA2: A PDA-BASED HIGHER-ORDER FLOW ANALYSIS FOR OBJECT-ORIENTED PROGRAMS

A Thesis
Presented to
The Academic Faculty

by

Nicholas Alexander Marquez

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
College of Computing

Georgia Institute of Technology
May 2013

OOCFA2: A PDA-BASED HIGHER-ORDER FLOW ANALYSIS FOR OBJECT-ORIENTED PROGRAMS

Approved by:

Professor Santosh Pande, Committee Chair
College of Computing
Georgia Institute of Technology

Professor Olin Shivers, Advisor
College of Computing and Information Science
Northeastern University

Professor Charles Isbell
College of Computing
Georgia Institute of Technology

Date Approved: January 9, 2013

To my parents;

For their unconditional emotional, financial, and spiritual support in my education and well-being; whatever that may entail and wherever it may take me

ACKNOWLEDGEMENTS

I would like the acknowledge the dedication and patience of my co-advisors Drs. Santosh Pande and Olin Shivers regarding the sisyphean task that was birthing this work. Without their guidance, I would be wandering aimlessly in my pursuit of an entrance into the Programming Language community and doctoral program therein. Indeed, I would likely have forgone pursuing PL and perhaps even academia long ago.

I would also like to acknowledge Drs. David Van Horn and Matt Might for their help and support in this endeavor. They've given me sage advice, helpful feedback, and much-needed concrete guidance where otherwise I would be (even more) hopelessly lost.

I would be remiss in leaving out (freshly) Dr. Dimitrios Vardoulakis for his invaluable aid in my understanding of—fundamentally—*his* algorithm, given that my work is largely a re-orienting of his CFA2 [25] [24].

Finally, I would like to thank my parents, my girlfriend Alexandria Llewellyn, and my friends for their kindness, understanding, and support in making sure I didn't fly *clear* off the sanity cliff. In particular I'd like to thank my parents for providing me comforting words and food, Alex for providing companionship and cakes, and my colleague Ian Johnson for providing perspective, help, and an attentive ear.

Contents

List of Tables

List of Figures

SUMMARY

The application of higher-order PDA-based flow analyses to object-oriented languages enables comprehensive and precise characterization of program behavior, while retaining practicality with efficiency. We implement one such flow analysis which we've named *OOCFA2*.

While over the years many advancements in flow analysis have been made, they have almost exclusively been with respect to functional languages, often modeled with the λ-calculus. Object-oriented semantics—while also able to be modeled in a functional setting—provide certain structural guarantees and common idioms which we believe are valuable to reason over in a first-class manner. By tailoring modern, advanced flow analyses to object-oriented semantics, we believe it is possible to achieve greater precision and efficiency than could be had using a functional modeling. This, in turn, reflects upon the possible classes of higher-level analyses using the underlying flow analysis: the more powerful, efficient, and flexible the flow analysis, the more classes of higher-level analyses—e.g., security analyses—can be practically expressed.

The growing trend is that smartphone and mobile-device (e.g., tablet) users are integrating these devices into their lives, in more frequent and more personal ways. Accordingly, the primary application and proof-of-concept for this work is the analysis of the Android [11] operating system's permissions-based security system vis-à-vis potentially malicious applications. It is implemented atop OOCFA2. The use of a such a powerful higher-order flow analysis allows one to apply its knowledge to create a wide variety of powerful and practical security-analysis "front-ends"—not only the permissions-checking analysis in this work, but also, e.g., information-flow analyses.

To our knowledge, OOCFA2 is the first higher-order flow analysis in an object-oriented setting. We empirically evaluate its accuracy and performance to prove its practical viability. We also evaluate the proof-of-concept security analysis' accuracy as directly

related to OOCFA2; this shows promising results for the potential of building security-oriented "front-ends" atop OOCFA2.

Chapter I

INTRODUCTION

1.1 Motivation

The growing trend is that smartphone and mobile-device (e.g., tablet) users are integrating these devices into their lives, in more frequent and more personal ways. Thus, users are exposing and even entrusting more sensitive information to devices that are becoming more and more omnipresent and laden with various applications, for example bank records, e-mail passwords, social-network credentials, sensitive files, and the contact list. There is a large amount of sensitive information available on a modern user's mobile device. Even further than the individual users themselves, these devices are also roaming, networked systems. This means that, for example, business networks can be compromised via a rogue, trusted device. Alternatively, a public or non-commercial network could be compromised to propagate the malware or to intercept all other users' transmissions.

There is a great amount and variety of real-world trouble that can result from mobile-device security violations. Accordingly, the primary application and proof-of-concept for this work is the analysis of the Android [11] operating system's permissions-based security system vis-à-vis potentially malicious applications.

While over the years many advancements in flow analysis have been made, they have almost exclusively been with respect to functional languages, often modeled with the λ-calculus. Object-oriented semantics—while also able to be modeled in a functional setting—provide certain structural guarantees and common idioms which we believe are valuable to reason over in a first-class manner. By tailoring modern, advanced flow analyses to object-oriented semantics, we believe it is possible to achieve greater precision and efficiency than could be had using a functional modeling. This, in turn, reflects upon the possible classes of higher-level analyses using the underlying flow analysis: the more powerful, efficient, and flexible the flow analysis, the more classes of higher-level analyses—e.g., security analyses—can be practically expressed.

1.2 High-level approach

The algorithm we describe in this work is a flow analysis using abstract interpretation. **Abstract interpretation** a static-analysis technique which involves computing a sound

approximation of the states in which a program may be. In a way, it's as if one is attempting to answer the question "what is the result of this program for all inputs?"—effectively "running" the program for all inputs. As it would be *at least* intractable and (more likely) uncomputable to attempt to, e.g., run a program for all integers between 0 and 2^{32}, abstract interpretation instead abstracts those values in some way. For our purposes, we consider it some unknown integer. (However some other algorithms will use domain-specialized representations, like the prefix domain for strings.) From this abstraction, it results that conditional control structures require that the algorithm explore all paths which are possible according to the set of possible inputs to the conditional. Thus, abstract interpretation ends up honestly exploring all possible paths through the program.

Of great consequence to the effectiveness of our algorithm is that the more precise an abstract-analysis algorithm is, the more likely it is that the algorithm will also take less time to analyze a program. This is because with greater precision there sometimes (depending on the means of the precision-increase) comes a decreased size in the abstract state space, and thus a decrease in the number of possible states the algorithm must explore. [18]

We apply this precision and efficiency to great effect on our motivating issue: Android security. We introduce in this work (chapter 4) a proof-of-concept permission-checking "front-end" whose precision and efficiency are directly tied to that of our underlying OOCFA2 algorithm.

1.3 Background: Android

(Unless otherwise noted, all discussion involving Android should be assumed to be referring to the latest Android version at the time of this writing: 4.1.1.)

Android is currently the world's most popular smartphone operating system, with a worldwide market share [27] of 68% at the second quarter of 2012. Furthermore, partly due to its use of the Linux kernel [17], Android has seen use in many areas outside the smartphone space, including netbooks, e-book readers, smart TVs, refrigerators, vehicle navigation systems, and more. Given such broad and personally-vested use, it is a very high-profile target for security exploitation.

It is built on the Linux kernel [17] (currently with out-of-tree customizations); a custom system C library called **bionic**; a variety of Android-specific middleware, libraries, and daemons; and a custom Java-running [20] virtual machine known as the **Dalvik virtual machine** [14] (**DVM**), with associated standard libraries based on Apache Harmony [6]. As a result of this composition, Android applications are typically written in C, C++, and Java, using the appropriate Android APIs and facilities.

In this section we discuss Android's security-oriented design decisions—in particular its permissions system. We also discuss the challenges these design decisions entail or leave un- or poorly-answered.

1.3.1 Android security

1.3.1.1 Overview

The Android operating system, development infrastructure, and larger ecosystem were designed with security in mind. Of course, the term "security" is heavily overloaded, especially in modern times, thus we refine this notion into Android's specific goals: [11]

- Protect user data
- Protect system resources (including, e.g., the cellular network)
- Protect application/author data, i.e., **DRM** (Digital Rights Management)

At a platform/system level, Android provides several high-level security features which further these goals: [11]

- Extensive use of kernel-provided security, permission, and isolation features
- Mandatory application sandboxing for all applications (e.g., using individual DVMs)
- Secure inter-process communication (**IPC**) for secure communication between isolated applications
- Application cryptographic signing
- Application- and system-defined and user-granted permissions

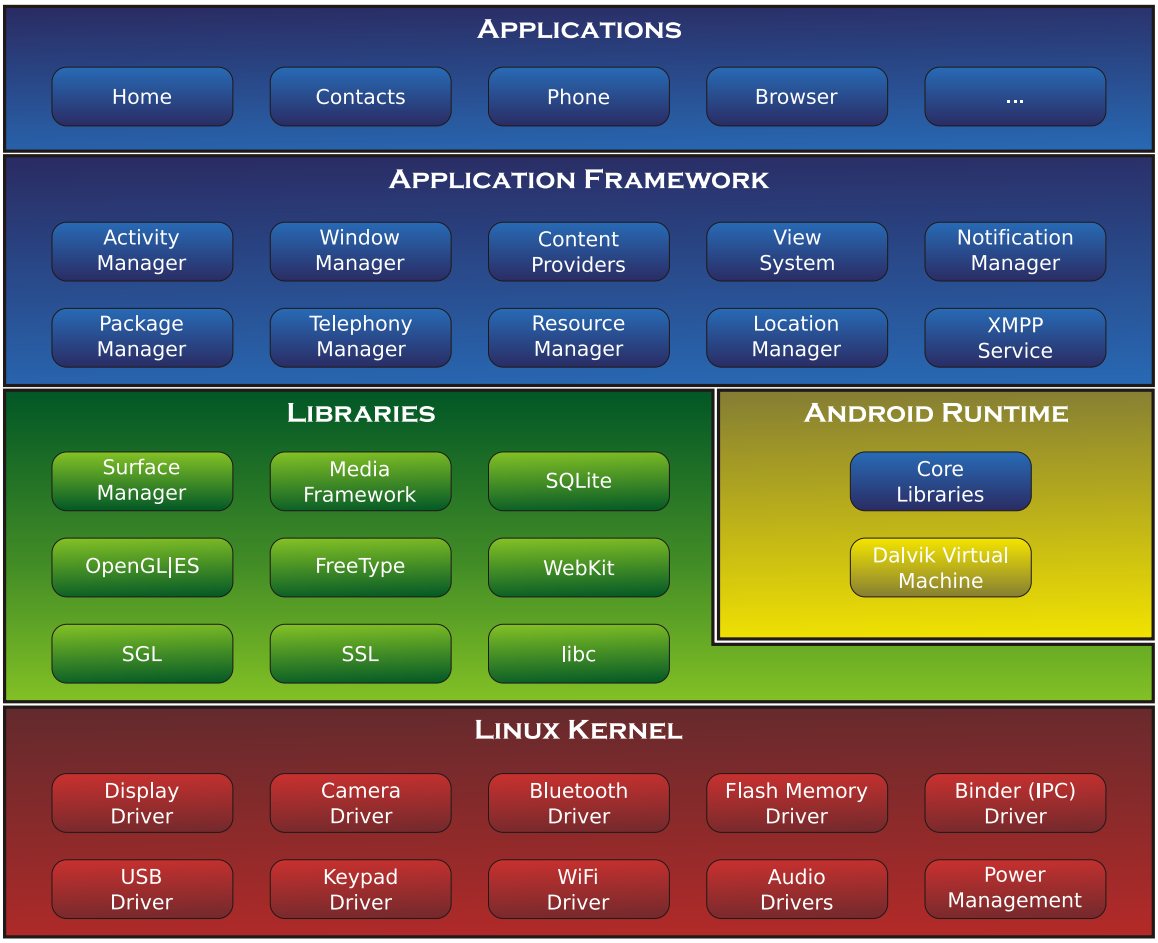

Figure 1: Android software stack [23]

More specifically, Android uses the Linux kernel's various security features along with relatively standard userspace solutions to achieve the following security enhancements:

- The system partition—containing the kernel, system libraries, application runtime and framework, and standard applications—is set to read-only
- Standard UNIX filesystem permissions are enforced; notably with user-differentiated applications (see below), this very effectively prevents undesired application data meddling
- Hardware-based No eXecute (**NX**) support, to prevent code execution on the stack and heap
- Use of OpenBSD's "dllmalloc" and "calloc" to protect against common programming errors and attacks involving heap allocation
- Robust whole filesystem encryption, using dm-crypt
- Position Independent Executable (**PIE**) support with Address Space Layout Randomization (**ASLR**) to randomize most memory locations
- Read-only relocations with link-time immediate binding (using the linker's "-z relro" and "-z now" options)
- Kernel addresses are not exposed in any way, given that "dmesg_restrict" and "kptr_restrict" are enabled
- Static protection against stack overflows (using gcc's "-fstack-protector" option)

Of particular note is that the Android system assigns a unique user ID (**UID**) to each application and runs it as that user in a separate process. This is in contrast to the typical operating systems approach of having multiple applications run under the same user permissions—of the user who runs them. In addition, each Dalvik application runs in its own dedicated Dalvik virtual machine, ensuring that this process-level and user-level isolation reaches non-native code. Because the applications are isolated in this way, they can only communicate under the purview of the kernel, using Android's provided IPC facilities.

1.3.1.2 Permissions

At the user-visible level, Android features an extensive—if ad-hoc—permissions-based security solution. These **permission**s are merely atomic predicates such as "the ability to access the network" or "the ability to modify the contact list". Notably, they cannot be combined into simple expressions (beyond simple conjunction) such as "the ability to access the network OR the ability to modify the contact list", nor more complex expressions such as "the ability to send the contact list over the network". Permissions also lack any notion of a "permissions lattice" or any other relations between them, such as "the ability to get a *fine-grained* location" being a sub-permission or otherwise implying "the ability to get a *coarse-grained* location".

When a user installs an application, she has the opportunity to review the permissions it requests. They can then make an informed choice of whether or not to install the application and thus grant it all such permissions; they cannot, however, grant or deny *individual* permissions. Combined with the deficiencies in the expressiveness of the permissions system, this means that a user must accept an application as requiring "the ability to send over the network" and "the ability to read the contact list" and simply trust that the application does not perform the higher-level, undesirable action of "reading the contact list and sending it over the network".

Permission checks are typically performed at Android API boundaries, where the API implementation performs the check in a system process. Therefore, not only are the permissions themselves ad-hoc and non-relational, the nature of the permissions checks is also ad-hoc. That is to say, permissions are checked in ad-hoc places throughout the Android APIs, with ad-hoc rules about re-checking. In particular, API library shims may perform the required permission-checking, but do so redundantly with the implementation, as an application could directly communicate with the underlying implementation by, e.g., communicating with the system process via an RPC stub. [9]

Some permissions are implemented not through Android's internal permission-validation mechanisms, but rather through the previously discussed use of standard Unix groups permissions. Notably, an application installed with the `INTERNET`, `BLUETOOTH`, or

WRITE_EXTERNAL_STORAGE permissions is assigned to a predefined Linux group which may access the relevant sockets and files related to those permissions.

Though Android allows (and in many situations, encourages) applications to include native code as well as DVM code, native code is still beholden to the permissions system. Permissions enforced via non-API means, such as the Unix permissions model, are still in effect for native code. Also, native code cannot directly interact with the system API; the application must create DVM wrappers to interact with the system APIs on behalf of the native code. In this manner, permissions-checking can focus on the Java-facing APIs.

In the context of this proof-of-concept application, we only target Android's permissions-based security system for analysis. I provide the following validation for this choice:

- It is a relatively contained and well-defined security system.
- It is the most user-facing security system.
- It is the most commonly exploited security system; not (necessarily) in terms of circumvention, but rather abusing the coarse, inexpressive permissions system, combined with user ignorance.

1.3.2 Challenges

The security and trustworthiness of applications deployed on mobile devices are both increasing important and difficult qualities to ensure. They carry sensitive data and have capabilities with significant social and financial effect. Yet, while it is paramount that such software is trustworthy, these applications pose challenges beyond the reach of current practice for low-cost, high-assurance verification and analysis. The domain is complex, involving interacting, event-driven programs; intricate security models; code from untrusted sources; highly sensitive data; and multiple channels on which data may be leaked.

Malicious agents are doing more and more to subvert and exploit these devices. Initially, this consisted of misusing permissions as granted by the Android security system, such as combining the ability to read the user's contact list with the ability to send over the network to thus send the user's contact list to a malicious third party. Nowadays, even the basic

permissions system as provided by Android is being circumvented in order to enact a whole array of new, undesirable behavior.

Typical approaches to combating malware, such as Google's *Bouncer* tool, involve cataloging known exploits and search codebases for patterns of misbehavior. This leads to an arms race between those exploiting the mobile environment and those fortifying it. We develop an alternative approach based on semantics-based program analysis that aims to bypass this race by giving a precise, sound, static characterization of mobile application behavior.

1.4 Contributions

This is the first PDA-based higher-order flow analysis in an object-oriented setting. With respect to the modeling of the Dalvik virtual machine, this is the first application of a higher-order flow analysis to an assembly language. We discuss the algorithm in chapter 3 and in particular our novel extensions to flow-analysis in section 3.3.

This work also features the first (published) formal model of the Dalvik virtual machine and its semantics. As far as we are aware, it is also most definitely the first formal model targeting Dalvik's ROP (cf. sections 2.1 and 2.2). We discuss this model—named Welterweight Dalvik—in section 2.3.

The Android-security-focused proof-of-concept application also is the first (published example) of its kind. We discuss this application in chapter 4. The use of such a powerful higher-order flow analysis allows one to apply its knowledge to create a wide variety of powerful and practical security-analysis "front-ends"—not only the permissions-checking analysis in this work, but also, e.g., information-flow analyses (subsubsection 6.1.3.1).

Chapter II

DALVIK OVERVIEW

The **Dalvik virtual machine** (**DVM**) is the primary and preferred means of running applications in the Android environment. It is a register-based, per-process virtual machine, thus emphasizing extreme application isolation and sandboxing to the point that every process runs in its own virtual machine instance.

In section 2.1 we discuss the DVM, the Dalvik toolchain, and where OOCFA2 fits in that toolchain. In section 2.2 we present the Dalvik ROP, which is the abstracted instruction-set for the DVM over which our algorithm reasons. We also present and discuss a formal model of the DVM and its runtime in section 2.3. Note that we assume of the reader a basic understanding of compilers and a very basic notion of operational semantics.

2.1 Dalvik Toolchain

The Android compilation toolchain is given in Figure 2. A Java compiler (noted in the figure as *javac*) takes the Android application's Java source, the Android libraries, and any other external Java libraries, and compiles and links them together to produce Java compiled class files. The **dx** tool is used to convert from Java compiled class files (whose instructions are JVM bytecode [19]) to Dalvik's **dex** bytecode format [5] (whose instructions are **DOP** [4]). The conversion process entails a simple "simulation" of a JVM execution of the JVM bytecode. It first converts from JVM bytecode to an intermediate representation, **ROP**. This is essentially a polymorphic and abstracted view of DOP. Most importantly, at this phase methods are internally represented as interconnected basic blocks. If optimization is enabled, dx then translates the ROP into an **SSA** (Static Single Assignment) form, upon which some simple optimizations are performed, and then back into normal ROP. From there, dx lowers from ROP to DOP, performs string and constant deduplication, and proceeds to pack all the resulting DOP, class information, etc. into the final dex file. Notably, the dx tool is the only officially-supported method of generating DOP, and is thus an integral part of the Android toolchain, making it the optimal target for extension.

2.1.1 OOCFA2 in the toolchain

OOCFA2 is integrated into the dx tool, thereby inserting itself as part of the Android development toolchain. It operates on the intermediate ROP form (non-SSA), though

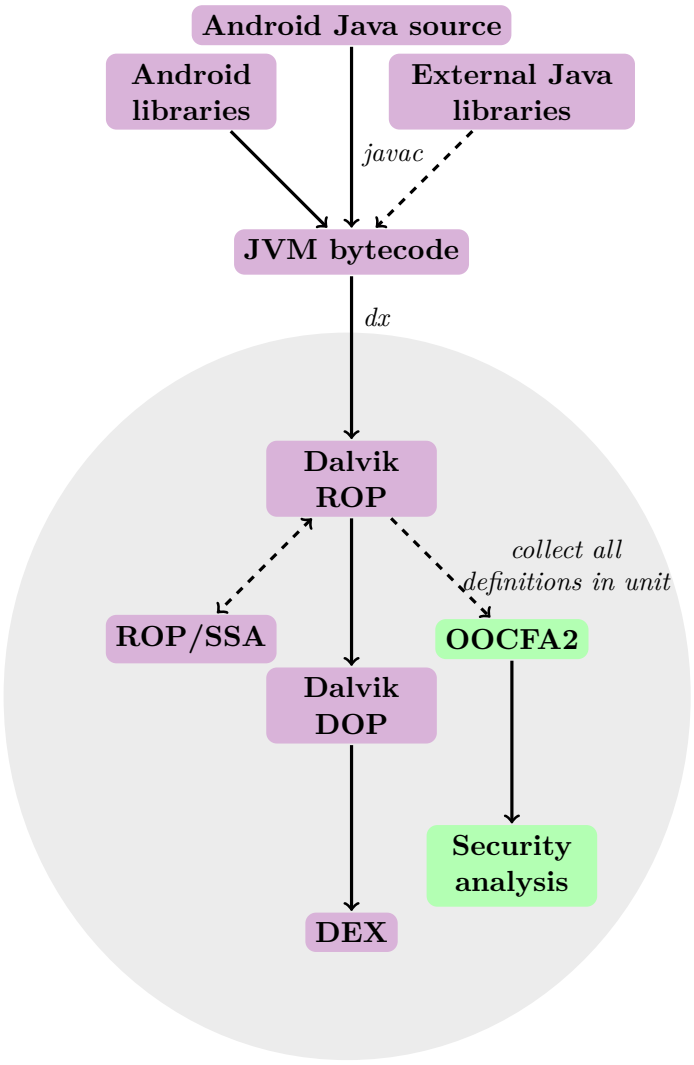

Figure 2: The Android compilation toolchain, through *dx*, including OOCFA2 additions

notably it does so simply to take advantage of ROP's polymorphism and relative abstraction; it would not be too difficult to have it operate upon DOP directly as part of future work. (Of note, this would allow OOCFA2 to reason over final dex files, possibly having been created or manipulated by hand or using external tools.)

The analysis passively collects the ROP forms of the entire compilation unit, then analyzes them and only afterward allows dx to continue and compile and output DOP. In this way, OOCFA2 and analyses using it can influence the intermediate ROP or prevent the creation of DOP altogether, depending on what the analyses conclude.

2.2 Dalvik ROP

The Dalvik ROP consists of instructions using the following opcodes:

- `NOP` is simply a no-op.
- `MOVE` copies a value from one register to another.
- `MOVE-PARAM` moves the next of a method's arguments into the given register.
- `MOVE-RESULT` moves the result of a method invocation into the given register. It may only be used following a method invocation.
- `MOVE-RESULT-PSEUDO` is the same as `MOVE-RESULT`, except that it must be used as the first instruction of a block following a non-invoke throwing instruction.
- `MOVE-EXCEPTION` moves a caught exception into the given register. It may only be used at the start of an exception handler.
- `CONST` moves the given constant value into the given register.
- `GOTO` jumps to the given label.
- `IF-EQ` tests the equality of two arguments and if they are equal jumps to the specified label. The arguments must both be integers or both be object (including arrays) addresses.

- `IF-NE` tests the inequality of two arguments and if they are not equal jumps to the specified label. The same rule as with `IF-EQ` applies.
- `IF-LT` tests whether one integer is less than the other and if so jumps to the specified label.
- `IF-GE` tests whether one integer is greater than or equal to the other and if so jumps to the specified label.
- `IF-LE` tests whether one integer is less than or equal to the other and if so jumps to the specified label.
- `IF-GT` tests whether one integer is greater than the other and if so jumps to the specified label.
- `SWITCH` indexes into the given jump-table to jump to a particular label.
- `ADD` adds two numbers.
- `SUB` subtracts two numbers.
- `MUL` multiplies two numbers.
- `DIV` divides two numbers.
- `REM` finds the remainder of the division of two numbers.
- `NEG` negates the given number.
- `AND` performs a bitwise-and on two integral numbers.
- `OR` performs a bitwise-or on two integral numbers.
- `XOR` performs a bitwise-exclusive-or on two integral numbers.
- `SHL` performs a bitwise-left-shift on two integral numbers.
- `SHR` performs a signed bitwise-right-shift on two integral numbers.
- `USHR` performs an unsigned bitwise-right-shift on two integral numbers.

- `NOT` performs a bitwise-negation of the given integral number.
- `CMPL` performs a Java-style "cmpl" on two numbers. This means that the result is 0 if both numbers are equal, 1 if the right is greater than the left, and -1 if the left is greater than the right. `NaN` is considered "less-than" all other values.
- `CMPG` performs a Java-tyle "cmpg" on two numbers. This is the same as with `CMPL`, except that `NaN` is considered "greater-than" all other values.
- `CONV` converts a number into one of the DVM's "real" numeric types: `int`, `long`, `float`, or `double`. The result is placed in the same register as the argument, effectively doing an in-place conversion. Which conversion is performed depends on the type of the result value.
- `TO-BYTE` converts the given integer into a byte (though the DVM stores it as a full-length integer) using a Java-style int-to-byte conversion. In pseudocode this would be $(i << 24) >> 24$.
- `TO-CHAR` is the same as `TO-BYTE` except that it uses a Java-style int-to-char conversion. In pseudocode this would be i & `0xffff`.
- `TO-SHORT` is the same as `TO-BYTE` except that it uses a Java-style int-to-short conversion. In pseudocode this would be $(i << 16) >> 16$.
- `RETURN` returns a value from a method, or nothing if the method's return-type is void.
- `THROW` throws the given exception in a method.
- `MONITOR-ENTER` enters the synchronization monitor for the given object.
- `MONITOR-EXIT` exits the synchronization monitor for the given object.
- `CHECK-CAST` checks whether the given object is an instance of the given reference-type specification; if not, then the DVM throws a `ClassCastException`.
- `INSTANCE-OF` also checks whether the given object is an instance of the given reference-type specification, results in an `int` 1 if true or an `int` 0 if false.

- **NEW-INSTANCE** allocates heap space for an object of the given class specification, in which it results.
- **NEW-ARRAY** allocates heap space for an array of the given element type specification and size, in which it results.
- **FILLED-NEW-ARRAY** is the same as **NEW-ARRAY**, though it initializes the given indices in the array with given values.
- **FILL-ARRAY-DATA** supplements **FILLED-NEW-ARRAY** with the data it is to use to fill the array.
- **ARRAY-LENGTH** results in the length of the given array.
- **AGET** gets the element at the given index in the given array.
- **APUT** puts the given value into the given index in the given array.
- **GET-FIELD** gets the instance field—according to the given instance field specification—of the given object.
- **GET-STATIC** gets the static field—according to the given static field specification.
- **PUT-FIELD** assigns the instance field—according to the given instance field specification—of the given object to the given value.
- **PUT-STATIC** assigns the static field—according to the given static field specification—to the given value.
- **INVOKE-STATIC** calls the static method—according to the given static method specification—with the given arguments.
- **INVOKE-VIRTUAL** calls the instance method—according to the given instance method specification and dispatched according to the given object—with the given arguments (and the given object prepended).

- `INVOKE-SUPER` is the same as `INVOKE-VIRTUAL`, but uses the given object's superclass for dispatch.
- `INVOKE-DIRECT` is the same as `INVOKE-VIRTUAL`, but calls a direct or special method.
- `INVOKE-INTERFACE` is the same as `INVOKE-VIRTUAL`, but uses an interface instance method specification.
- `MARK-LOCAL` is simply for a debugger's use; it marks the name of a local variable.

2.3 Welterweight Dalvik: Formal Model of DVM/ROP

In this section, we describe our formal model of the DVM given ROP: **Welterweight Dalvik**. We give the grammar of the language's syntax in Figure 3, define our formal conceptualization of the ROP instructions in Figure 4, and the semantic domains of our abstract machine model in Figure 5. The reduction rules for the abstract machine are presented in subsection 2.3.2, interspersed with prose presenting them.

Welterweight Dalvik was developed using Redex: a domain-specific language for specifying reduction semantics, plus a suite of tools for working with the semantics. [10] A formal model of OOCFA2 was likewise developed, but as it currently has a soundness issue, we have excluded it from this work. We have tested and compared both models using a small suite of custom-made programs (written in the Welterweight Dalvik grammar) designed to stress particular aspects of the semantics and OOCFA2 algorithm.

Welterweight Dalvik is valuable because Dalvik was not designed with a formal semantics in mind and thus it is impossible to prove various properties of the system. Likewise, our formal model of OOCFA2 is essential to prove various properties—most importantly among them, soundness—and especially vis-à-vis the concrete semantics.

2.3.1 Grammar

We define the grammar of the language's syntax in Figure 3.

The top level of a program is given by *Top-Level* and is simply a nonempty list of class definitions followed by the identification of which static method should be considered the "main" function.

Top-Level ::= [*Class* ...$_+$ (`main` *FQ-Method-Spec*)]
Class ::= (`class` *Class-Spec*
Visibility [*Attr* ...]
Super
[*Field* ...]
[*Method* ...])
Super ::= *FQ-Class-Spec* | `⊤`
Visibility ::= `public` | `protected` | `private`
Attr ::= `static` | `final`
Field ::= (`field` *Field-Spec*
Visibility [*Attr* ...]
Type lit)
Method ::= (`method` *Method-Spec*
Visibility [*Attr* ...]
[*Type* ...] *Ret-Type*
Stmt ...$_+$)
Ret-Type ::= *Type* | `void`
Type ::= *Prim-Type* | *Ref-Type*
Prim-Type ::= `int`
Ref-Type ::= *FQ-Class-Spec* | *Array-Type*
Array-Type ::= (`A` *Type*)
Spec ::= *string* conforming to Java identifier naming
FQ-Spec ::= *string* conforming to Java fully-qualified-identifier naming
lit ::= *i* | *o* | *a* | `null`
i ::= *integer*
o ::= (`obj` *FQ-Class-Spec IFields*)
IFields = (*Field-Spec* → *v*)
v ::= *i* | *addr* | `null`
addr ::= (`addr` *integer*)
a ::= (`array` *Type Size Elems*) given $\nexists i \in Elems \mid i \geq Size$
Size ::= *integer* given $0 \leq integer < 256$
Elems = (*Size* → *v*)
Stmt ::= *Instruction* | (`label` *Label*) | *TCF* | `halt`
Label ::= *string*
TCF ::= (`try` [*Instruction* ...$_+$]
`catch` [*Instruction* ...]
`finally` [*Instruction* ...])

Figure 3: Grammar of Welterweight Dalvik

A *Class* is a syntactic definition of a Java class, complete with its specification *Class-Spec*, visibility *Visibility*, attributes *Attr*..., superclass *Super*, fields [*Field* ...], and methods [*Method* ...].

A *Field* is a syntactic definition of a Java field, complete with its specification *Field-Spec*, its visibility and attributes as before, its type *Type*, and its initial (default) value *lit*.

A *Method*is a syntactic definition of a Java method, complete with its specficiation *Method-Spec*, its visibility and attributes as before, its formal parameter types [*Type* ...], its return type *Ret-Type*, and its body *Stmt* ...$_+$.

A return type may be any *Type* or `void`. A *Type* is either a primitive type *Prim-Type* or a reference type *Ref-Type*. The only primitive type in Welterweight Dalvik is `int`. A *Ref-Type* is either denoted by a full class specification *FQ-Class-Spec*, or is an array type *Array-Type*. An *Array-Type*, in turn, is simply a syntactically distinguished embedding of the element *Type* of the array.

Specifications are listed in the grammar under the syntactic template *Spec*. They are simply strings which follow the identifier-naming rules of Java. A full specification is denoted by the convention of prefixing a specification with `FQ-` in the grammar. It is a string which simply describes a dot-separated path of specifications, where all but the last specification must be *Class-Spec*s.

Literals are given by *lit* and are either integers (i), objects (o), arrays (a), or the null pointer `null`. An o is a syntactic definition of an object, complete with the specification of its class (*FQ-Class-Spec*) and its instance fields (*IFields*). *IFields* is a map from *Field-Spec* to values v. In turn, vs represent run-time, user-accessible values, which are either is, addresses (*addr*), or `null`. An *addr* is simply a syntactically distinguished integer representing a heap address. An a is a syntactic definition of an array, complete with the type of its elements (*Type*), its size (*Size*), and its elements (*Elems*), where each element index is within the array's size. A *Size* is simply a bounded integer. *Elems* is a map from indices (given by *Size* to imply the bound) to the elements themselves: vs.

In particular, literal objects and arrays are really only user-accessible as a convenient means of denoting complex initial state in a program; as we shall see, when loaded into

the machine model, they are indirected and replaced with vs denoting their address in the initial program heap.

A *Stmt* is either an instruction (*Instruction*), a label definition, a try-catch-finally block (*TCF*), or the special halt instruction `halt`. A *Label* is simply a string used to identify a method-local label. A *TCF* is a try-catch-finally block where the set of instructions composing each part of the block is given by a [*Instruction* ...]. The `try` portion of a *TCF* cannot be empty, though the other parts can be.

Instruction	::=	`(nop)`
	|	`(move` *r r*`)`
	|	`(move-param` *r*`)`
	|	`(move-result` *r*`)`
	|	`(move-exception` *r*`)`
	|	`(const` *r i*`)`
	|	`(goto` *Label*`)`
	|	`(`*If r r Label*`)`
	|	`(`Op_2 *r r r*`)`
	|	`(`Op_1 *r r*`)`
	|	`(cmp` *r r r*`)`
	|	`(`*Conv r*`)`
	|	`(return` *r*`)`
	|	`(return-void)`
	|	`(throw` *r*`)`
	|	`(new-instance` *r FQ-Class-Spec*`)`
	|	`(new-array` *r Type r*`)`
	|	`(check-cast` *r Ref-Type*`)`
	|	`(instance-of` *r r Ref-Type*`)`
	|	`(array-len` *r Ref-Type*`)`
	|	`(aget` *r r r*`)`
	|	`(aput` *r r r*`)`
	|	`(iget` *r r Field-Spec*`)`
	|	`(iput` *r r Field-Spec*`)`
	|	`(sget` *r Field-Spec*`)`
	|	`(sput` *r Field-Spec*`)`
	|	`(invoke-virtual` *Method-Spec r r* ...`)`
	|	`(invoke-super` *Method-Spec r r* ...`)`
	|	`(invoke-static` *Method-Spec r* ...`)`
If	::=	`if-eq | if-ne | if-ge | if-le | if-gt`
Op_2	::=	`add | sub | mul | div | rem`
	|	`and | or | xor | shl | shr | ushr`
Op_1	::=	`neg | not`
Conv	::=	`to-byte | to-char | to-short`
r	::=	register variable with $-prefix

Figure 4: Instructions in Welterweight Dalvik

We define our grammar of the ROP instructions in Figure 4.

We will not discuss every instruction individually, as they largely reflect the actual DVM ROP opcodes and as the reduction rules (subsection 2.3.2) should give clear definitions of the instructions. However, we will highlight some of the choices made in reducing DVM ROP into Welterweight Dalvik:

- We ignore the superfluous `MOVE-RESULT-PSEUDO` opcode.
- The ROP actually has a variant of the `CONST` instruction which loads a class specification into a register. We instead choose to directly encode class-specification constants into any instruction that requires them.
- The `SWITCH` opcode is irrelevant for modeling purposes, as it can be trivially desugared into chained conditional tests.
- Monitors are immaterial to our modeling of the Dalvik system; multi-threading is a difficult problem for traditional flow analyses and it is outside the scope of this work (cf. subsubsection 3.2.2.2). We therefore exclude the `MONITOR-ENTER` and `MONITOR-EXIT` opcodes.
- Filled arrays are also immaterial to our model, though for convenience's sake we do allow the user to declare filled arrays as initial values. We therefore exclude the `FILLED-NEW-ARRAY` and `FILLED-ARRAY-DATA` opcodes.
- `INVOKE-DIRECT` is simply used as an optimization (when the virtual method involved can be statically identified) or for invoking native code through Java (using the *JNI*). As we don't care about optimizations and we don't attempt to model native code interaction, we thus treat `INVOKE-DIRECT` as subsumed by `invoke-virtual`.
- `INVOKE-INTERFACE` is likewise vestigial; Dalvik uses it to indirect through an interface when performing a virtual method call. As we don't attempt to model Java's interface semantics (i.e., multiple inheritance in general), we don't ever do interface indirect in the model. Therefore we also treat `INVOKE-INTERFACE` as subsumed by `invoke-virtual`.

$$
\begin{array}{rcl}
\mathscr{C} & = & (\textit{FQ-Class-Spec} \rightarrow \textit{Class}) \\
\mathscr{M} & = & (\textit{FQ-Method-Spec} \rightarrow \textit{Method}) \\
\mathscr{F} & = & (\textit{FQ-Field-Spec} \rightarrow \textit{Field}) \quad \text{where all } \textit{lit} \text{ are transformed to } v \\
\mathscr{P} & ::= & \langle \mathbb{I}, \mathbb{R}, \mathbb{S}, \mathbb{N}, \mathbb{E}, \mathbb{H}, \mathbb{Z}, \mathbb{M} \rangle \\
\mathbb{I} & ::= & \varepsilon \mid \textit{Stmt}\,\mathbb{I} \mid \texttt{handle} \\
\textit{Instruction} & ::= & \ldots. \mid (\texttt{invoke}\ \textit{Method}\ r \ldots) \mid (\texttt{return*}\ h_r) \\
\mathbb{R} & ::= & \varepsilon \mid \boldsymbol{R}\,\mathbb{R} \quad \text{where} \quad \boldsymbol{R} ::= (r \rightarrow v) \\
\mathbb{S} & ::= & \varepsilon \mid \boldsymbol{S}\,\mathbb{S} \quad \text{where} \quad \boldsymbol{S} ::= \varepsilon \mid v\,\boldsymbol{S} \\
\mathbb{N} & ::= & \varepsilon \mid \boldsymbol{N}\,\mathbb{N} \quad \text{where} \quad \boldsymbol{N} ::= \varepsilon \mid \textit{Stmt}\,\boldsymbol{N} \\
\mathbb{E} & ::= & \varepsilon \mid \boldsymbol{E}\,\mathbb{E} \quad \text{where} \quad \boldsymbol{E} ::= \varepsilon \mid \textit{Stmt}\,\boldsymbol{E} \\
\mathbb{H} & = & (\textit{addr} \rightarrow \textit{Ref}) \cup (\textit{FQ-Field-Spec} \rightarrow \textit{Ref}) \\
\textit{Ref} & ::= & o \mid a \\
\mathbb{Z} & ::= & \langle h_r, h_e \rangle \\
h_* & ::= & v \mid \bot \\
\mathbb{M} & ::= & \varepsilon \mid \textit{Method}\,\mathbb{M}
\end{array}
$$

Figure 5: Machine model for Welterweight Dalvik

We define the grammar used in our abstract machine model in Figure 5.

The Welterweight Dalvik machine model is based centrally on four elements: the class registry ($\mathscr{C}$), the static-method registry ($\mathscr{M}$), the static-field registry ($\mathscr{F}$), and the program state ($\mathscr{P}$). The registries are simply maps from full specifications to their respective program elements; they are prepopulated outside the purview of the machine's reduction model itself, using the information provided by the program (through the *Top-Level*). The $\mathscr{P}$ is what is interesting: it is the element which is transformed during a reduction rule in the model. It is represented by a record composed of the following elements:

- An instruction stream $\mathbb{I}$. It is represented as a string which is either empty, has a *Stmt* on top, or is the special token `handle`. For the convenience of the machine model, we add an additional two pseudo-instructions: `invoke` and `return*`.

- A register stack $\mathbb{R}$. We use a stack to help enforce the notion that methods—even in the DVM (vs. the JVM)—are not supposed to communicate through the registers, but rather using the parameter and return stacks. Thus a method is confined to using only its set of registers. It is represented as a string which is either empty or has a $\boldsymbol{R}$ on top. A $\boldsymbol{R}$ is, in turn, a map from registers r to values v.

- A nested parameter stack $\mathbb{S}$. This is the stack of stacks of parameters passed to

methods. Like $\mathbb{R}$, it is a stack (in this case nested) because methods are not supposed to be able to access the parameters of an earlier caller; thus a method can only access the parameter stack intended for it. It is represented as a string which is either empty or has a $\boldsymbol{S}$ on top. A $\boldsymbol{S}$ is, in turn, the actual (non-nested) parameter stack for a method. It too is represented as a string which is either empty or has a v on top.

- A return stack $\mathbb{N}$. This is the stack of return points for method invocations. It is represented as a string which is either empty or has a $\boldsymbol{N}$ on top. A $\boldsymbol{N}$ in turn is simply the instruction stream to which to return, represented as a string which is either empty or has a *Stmt* on top.

- An exception-handler stack $\mathbb{E}$. This is the stack of exception-handling catch-points for method invocations. It is represented as a string which is either empty or has a $\boldsymbol{E}$ on top. A $\boldsymbol{E}$ is has the same purpose as a $\boldsymbol{N}$ and is represented identically.

- A heap $\mathbb{H}$. This is the heap which indirects all referential data in the program. It is a map to referential data *Ref*, either from addresses *addr* or from full static field specifications *FQ-Field-Spec*. A *Ref* is simply either an object o or an array a. Outside the purview of the reduction model, it is prepopulated with the initial values of static fields.

- A pair of hidden registers $\mathbb{Z}$. It represents the odd hidden registers in the DVM ROP machine model which are used to store a method's return value (h_r) or a method's thrown exceptional (h_e). It is represented as a record of those two registers. A hidden register is either full with some value v or empty, denoted using $\perp$.

- A stack trace of the call-chain of methods $\mathbb{M}$. It is represented as a string which is either empty or has a *Method* on top. It is used in order to identify the current method, e.g., for label-resolution purposes.

2.3.2 Reduction rules

In this subsection we give the reduction rules for the machine model of Welterweight Dalvik and summarize them in prose. If no rule matches—in particular if an instruction-handling rule would match but one of its side-conditions if violated—then the program reduction reaches a **stuck state**, indicating a semantically invalid program. In particular, in some situations the DVM (and JVM) would throw an exceptional deriving from `RuntimeException` or `Error`—collectively known as **unchecked exceptionals**. The Welterweight Dalvik model instead treats these conditions as stuck states. Validation for this can be found in the discussion of OOCFA2's design choices (subsubsection 3.2.2.1).

Each reduction rule takes the form of a judgment from premises to conclusions, split by a horizontal line. The reduction is in term of the entire program state $\mathscr{P}$, using the relation $\mathscr{P} \hookrightarrow \mathscr{P}$. As $\mathscr{P}$ is a record, we take some notational liberty for clarity and conciseness. Namely, the relation $\hookrightarrow$ may be parameterized by certain machine/program elements. This explicitly denotes that they are used but preserved during the relation, i.e., the noted element does is not itself transformed during the relation, though it may be part of a larger transformation. See Figure 6 for an example of the of a reduction rule and its long-form expansion. For the sake of clarity, we also make sure to explicitly note in the premise any program elements which will be used in the relation, even if they are wholly preserved.

$$\underset{R}{\overset{\text{OP}_1}{\hookrightarrow}} \frac{\mathscr{P}\,\langle\,(Op_1\ r_{to}\ r_{arg})\,\mathbb{I} \qquad (R[r_{arg} \to i_{arg}])\,\mathbb{R}\rangle}{\mathscr{P}\,\langle\,\mathbb{I} \qquad (R[r_{to} := i_{ret}])\,\mathbb{R}\rangle}$$
$$\text{where} \quad i_{ret} = \delta_1(Op_1,\ i_{arg})$$

(a) Short-form example

$$\underset{R}{\overset{\text{OP}_1}{\hookrightarrow}} \frac{\mathscr{P}\,\langle\,(Op_1\ r_{to}\ r_{arg})\,\mathbb{I} \qquad (R[r_{arg} \to i_{arg}])\,\mathbb{R}\rangle}{\mathscr{P}\,\langle\,\mathbb{I} \qquad (R[r_{arg} \to i_{arg},\ r_{to} := i_{ret}])\,\mathbb{R}\rangle}$$
$$\text{where} \quad i_{ret} = \delta_1(Op_1,\ i_{arg})$$

(b) Long-form expansion

Figure 6: Welterweight Dalvik reduction rule example

$$\hookrightarrow \frac{\text{LABEL}\quad \mathscr{P}\,\langle (\texttt{label}\ \mathit{Label})\,\mathbb{I}\rangle}{\mathscr{P}\,\langle \mathbb{I}\rangle}$$

$$\hookrightarrow \frac{\text{NOP}\quad \mathscr{P}\,\langle (\texttt{nop})\,\mathbb{I}\rangle}{\mathscr{P}\,\langle \mathbb{I}\rangle}$$

(a) No-ops

Figure 7: Welterweight Dalvik reduction rules

These rules are no-ops. The LABEL rule exists simply so that if we reach a `goto`, we have a means by which to identify what point in an instruction stream to which to jump.

$$\hookrightarrow \frac{\text{TRY-CATCH-FINALLY}\quad \mathscr{P}\,\langle \mathbb{I} : (\texttt{try}\ [\mathit{Stmt}_{try}\ldots]\ \texttt{catch}\ [\mathit{Stmt}_{catch}\ldots]\ \texttt{finally}\ [\mathit{Stmt}_{finally}\ldots])\,\mathbb{I}' \qquad \mathbb{E}\rangle}{\mathscr{P}\,\langle \mathbb{I} : [\mathit{Stmt}_{try}\ldots\ \mathit{Stmt}_{finally}\ldots]\,\mathbb{I}' \qquad ((\mathit{Stmt}_{catch}\ldots\ \mathit{Stmt}_{finally}\ldots)\,\mathbb{I}')\,\mathbb{E}\rangle}$$

(b) Try-Catch-Finally

Figure 7: Welterweight Dalvik reduction rules

The TRY-CATCH-FINALY rule is actually a syntactic convenience for Welterweight Dalvik programs. The DVM itself does not have a notion of ***TCF*** or of scope, and thus there it is desugared into `goto`s and manual exception-handling. In keeping with the notion of this simply being a syntactic convenience, the model does not attempt to emulate scope and thus, e.g., a ***TCF*** statement cannot embed `goto`s (otherwise the semantics of `finally` would be broken). Furthermore, a `catch` block is responsible for rethrowing any exceptions which it does not handle, as is the case in the DVM-proper.

$$\hookrightarrow \frac{\text{MOVE}\quad \mathscr{P}\,\langle (\texttt{move}\ r_{to}\ r_{from})\ \mathbb{I} \qquad R\,\mathbb{R}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := v])\,\mathbb{R}\rangle}$$
where $\quad v = R(r_{from})$

$$\hookrightarrow \frac{\text{MOVE-PARAM}\quad \mathscr{P}\,\langle (\texttt{move-param}\ r_{to})\ \mathbb{I} \qquad R\,\mathbb{R} \qquad (v\,S)\,\mathbb{S}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := v])\,\mathbb{R} \qquad S\,\mathbb{S}\rangle}$$

$$\hookrightarrow \frac{\text{MOVE-RESULT}\quad \mathscr{P}\,\langle (\texttt{move-result}\ r_{to})\ \mathbb{I} \qquad R\,\mathbb{R} \qquad \mathbb{Z}\langle h_r : v\rangle\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := v])\,\mathbb{R} \qquad \mathbb{Z}\langle h_r : \bot\rangle\rangle}$$

$$\hookrightarrow \frac{\text{MOVE-EXCEPTION}\quad \mathscr{P}\,\langle (\texttt{move-exception}\ r_{to})\ \mathbb{I} \qquad R\,\mathbb{R} \qquad \mathbb{Z}\langle h_e : v\rangle\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := v])\,\mathbb{R} \qquad \mathbb{Z}\langle h_e : \bot\rangle\rangle}$$

$$\hookrightarrow \frac{\text{CONST}\quad \mathscr{P}\,\langle (\texttt{const}\ r_{to}\ i)\ \mathbb{I} \qquad R\,\mathbb{R}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := i])\,\mathbb{R}\rangle}$$

(c) Movement

Figure 7: Welterweight Dalvik reduction rules

These rules handle movement of values into registers.

- The MOVE rule copies a value from one register r_{from} to another $\mathbb{R}_{to}$.
- The MOVE-PARAM rule pops a value v from the parameter stack $\mathbb{S}$ and puts it in the destination register r_{to}.
- The MOVE-RESULT rule moves the return value v stored in the hidden register h_r into the destination register r_{to}.
- The MOVE-EXCEPTION rule does the same, but for exception values, using the hidden register h_e.
- The CONST rule directly loads a constant i into the destination register r_{to}.

$$\hookrightarrow_R \frac{\text{OP}_2 \quad \mathscr{P}\,\langle ((Op_2\ r_{to}\ r_{arg1}\ r_{arg2})\ \mathbb{I} \qquad (R[r_{arg1} \to v_1,\ r_{arg2} \to v_2])\,\mathbb{R}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := i_{ret}])\,\mathbb{R}\rangle}$$
where $\quad i_{ret} = \delta_2(Op_2,\ i_1,\ i_2)$

$$\hookrightarrow_R \frac{\text{OP}_1 \quad \mathscr{P}\,\langle ((Op_1\ r_{to}\ r_{arg})\ \mathbb{I} \qquad (R[r_{arg} \to i_{arg}])\,\mathbb{R}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := i_{ret}])\,\mathbb{R}\rangle}$$
where $\quad i_{ret} = \delta_1(Op_1,\ i_{arg})$

$$\hookrightarrow_R \frac{\text{CMP} \quad \mathscr{P}\,\langle (\texttt{cmp}\ r_{to}\ r_{arg1}\ r_{arg2})\ \mathbb{I} \qquad (R[r_{arg1} \to i_1,\ r_{arg2} \to i_2])\,\mathbb{R}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := i_{ret}])\,\mathbb{R}\rangle}$$
where $\quad i_{ret} = \delta_{cmp}(i_1,\ i_2)$

$$\hookrightarrow \frac{\text{CONV} \quad \mathscr{P}\,\langle ((Conv\ r_{arg/to})\ \mathbb{I} \qquad (R[r_{arg/to} \to i_{arg}])\,\mathbb{R}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{arg/to} := i_{ret}])\,\mathbb{R}\rangle}$$
where $\quad i_{ret} = \delta_{conv}(Conv,\ i_{arg})$

(d) Operations

Figure 7: Welterweight Dalvik reduction rules

These rules handle various primitive operations on integers.

- The OP$_2$ rule performs a binary operation on the integers within registers r_{arg1} and r_{arg2}, using the metafunction δ_2 to perform the operation; it stores the result in the destination register r_{to}.
- The OP$_1$ rule behaves similarly, but for unary operations using register r_{arg} and the metafunction δ_1, which performs the operation.
- The CMP rule behaves similarly to OP$_2$. However, it uses the metafunction δ_{cmp}, which compares the given integers and returns -1, 0, or 1 depending on if the first integer is less than, equal to, or greater than the second integer, respectively.
- The CONV rule behaves similarly to OP$_1$, but the destination register and argument register are the same: $r_{arg/to}$. It also instead uses the metafunction δ_{conv}, which truncates an integer to the requested bit-width.

$$\underset{\mathbb{R},\,\mathbb{M}}{\hookrightarrow} \frac{\textsc{If}\quad \mathscr{P}\,\langle (\mathit{If}\ r_1\ r_2\ \mathit{Label})\,\mathbb{I} \qquad (\mathit{R}[r_1 \to v_1, r_2 \to v_2])\,\mathbb{R} \qquad \mathit{Method}\,\mathbb{M}\rangle}{\mathscr{P}\,\langle \mathbb{I}'\rangle}$$

where $\mathbb{I}' = \begin{cases} \delta_{goto}(\mathit{Label},\ \mathit{Method}) & \text{if} \quad \delta_{if}(\mathit{If},\ v_1,\ v_2) \\ \mathbb{I} & \text{otherwise} \end{cases}$

$$\underset{\mathbb{M}}{\hookrightarrow} \frac{\textsc{goto}\quad \mathscr{P}\,\langle (\texttt{goto}\ \mathit{Label})\,\mathbb{I} \qquad \mathit{Method}\,\mathbb{M}\rangle}{\mathscr{P}\,\langle \mathbb{I}'\rangle}$$

where $\mathbb{I}' = \delta_{goto}(\mathit{Label},\ \mathit{Method})$

(e) Branches

Figure 7: Welterweight Dalvik reduction rules

These rules handle primitive branching to *Label*s.

- The If rule uses the values in argument registers r_1 and r_2 to evaluate whether to branch to *Label* according to the metafunction δ_{if}; the actual instruction stream to which to branch is found using the δ_{goto} metafunction.
- The goto rule simply unconditionally performs this branch.

$$\hookrightarrow_{\mathbb{R}} \frac{\text{RETURN}\quad \mathscr{P}\langle(\texttt{return}\ r)\ \mathbb{I} \qquad (R[r \to v_{ret}])\,\mathbb{R}\rangle}{\mathscr{P}\langle(\texttt{return*}\ v_{ret})\ \mathbb{I}\rangle}$$

$$\hookrightarrow \frac{\text{RETURN-VOID}\quad \mathscr{P}\langle(\texttt{return-void})\ \mathbb{I}\rangle}{\mathscr{P}\langle(\texttt{return*}\ \bot)\rangle}$$

$$\hookrightarrow \frac{\text{RETURN}^*\quad \mathscr{P}\langle(\texttt{return*}\ h_r)\ \mathbb{I} \qquad R\,\mathbb{R} \qquad \mathbb{I}'\,\mathbb{N} \qquad E\,\mathbb{E} \qquad \mathbb{Z} \qquad Method\,\mathbb{M}\rangle}{\mathscr{P}\langle\mathbb{I}' \qquad \mathbb{R} \qquad \mathbb{N} \qquad \mathbb{E} \qquad \mathbb{Z}\langle h_r\rangle \qquad \mathbb{M}\rangle}$$

(f) Returning

Figure 7: Welterweight Dalvik reduction rules

These rules handle returning from methods.

- The RETURN rule simply transitions to a `return*`, using the value in the given register r as the operand.
- The RETURN-VOID rule likewise transitions to a `return*`, but uses $\bot$ as the operand.
- The RETURN* rule is the actual workhorse for returning; places its operand into the hidden register h_r, pops off all method-scoped stacks, and returns control to the suspended caller on top of the $\mathbb{N}$.

$$\hookrightarrow_{\mathbb{R}} \ \text{THROW}\ \frac{\mathscr{P}\,\langle (\texttt{throw}\ r_{exn})\,\mathbb{I} \qquad (R[r_{exn} \to v_{exn}])\,\mathbb{R} \qquad \mathbb{Z}\rangle}{\mathscr{P}\,\langle \mathbb{I} : \texttt{handle} \qquad \mathbb{Z}\,\langle h_e : v_{exn}\rangle\rangle}$$

$$\hookrightarrow_{\mathbb{I},\,\mathbb{Z}} \ \text{HANDLE-PROPAGATE}\ \frac{\mathscr{P}\,\langle \mathbb{I} : \texttt{handle} \qquad R\,\mathbb{R} \qquad N\,\mathbb{N} \qquad []\,\mathbb{E} \qquad \mathbb{Z}\,\langle h_r : v\rangle\, Method\,\mathbb{M}\rangle}{\mathscr{P}\,\langle \mathbb{R} \qquad \mathbb{N} \qquad \mathbb{E} \qquad \mathbb{M}\rangle}$$

$$\hookrightarrow_{\mathbb{Z}} \ \text{HANDLE}\ \frac{\mathscr{P}\,\langle \mathbb{I} : \texttt{handle} \qquad R\,\mathbb{R} \qquad N\,\mathbb{N} \qquad \mathbb{I}'\,\mathbb{E} \qquad \mathbb{Z}\,\langle h_r : v\rangle \qquad Method\,\mathbb{M}\rangle}{\mathscr{P}\,\langle \mathbb{I}' \qquad \mathbb{R} \qquad \mathbb{N} \qquad \mathbb{E} \qquad \mathbb{M}\rangle}$$

(g) Throwing/Handling

Figure 7: Welterweight Dalvik reduction rules

These rules deal with the throwing, propagation, and handling of exceptions.

- The THROW rule places the value in the given register r_{exn} into the hidden register h_e; it then begins the process of handling the exception by setting the instruction stream $\mathbb{I}$ to the special token `handle`.
- The HANDLE-PROPAGATE rule is triggered if the top of the $\mathbb{E}$ is empty; in this case we pop off all method-scoped stacks and continue to attempt to `handle` the exception.
- The HANDLE rule is triggered if the top of the $\mathbb{E}$ holds a proper handler; in this case we pop off all method-scoped stacks and jump to the handler's code.

$$\hookrightarrow \frac{\text{NEW-INSTANCE}\quad \mathscr{P}\,\langle (\texttt{new-instance}\ r_{to}\ \textit{FQ-Class-Spec})\,\mathbb{I} \qquad R\,\mathbb{R} \qquad \mathbb{H}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := addr_{new}])\,\mathbb{R} \qquad \mathbb{H}[addr_{new} := o_{new}]\rangle}$$

where $addr_{new} \notin \mathbb{H}$

where $o_{new} = (\texttt{obj}\ \textit{FQ-Class-Spec}\ \varepsilon)$

$$\underset{R}{\hookrightarrow} \frac{\text{NEW-ARRAY}\quad \mathscr{P}\,\langle (\texttt{new-array}\ r_{to}\ \textit{Type}\ r_{size})\,\mathbb{I} \qquad (R[r_{size} \to \textit{Size}])\,\mathbb{R} \qquad \mathbb{H}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := addr_{new}])\,\mathbb{R} \qquad \mathbb{H}[addr_{new} := a_{new}]\rangle}$$

where $addr_{new} \notin \mathbb{H}$

where $a_{new} = (\texttt{array}\ \textit{Type}\ \textit{Size}\ \varepsilon)$

(h) Allocation

Figure 7: Welterweight Dalvik reduction rules

These rules handle the raw allocation of new reference types. Note that this does not involve initialization—i.e., an object's class's constructors. Constructors must be manually emulated in Welterweight Dalvik.

- The NEW-INSTANCE rule allocates a new address on the heap, stores there a new object of the given class *FQ-Class-Spec*, and returns the address through the destination register r_{to}.
- The NEW-ARRAY rule behaves similarly to NEW-INSTANCE, but stores a new array of the given type *Type* and size *Size*.

$$\overset{\text{CHECK-CAST}}{\underset{\mathbb{R},\,\mathbb{H}}{\hookrightarrow}} \frac{\mathscr{P}\,\langle (\texttt{check-cast}\ r_{of}\ \textit{Ref-Type})\,\mathbb{I} \qquad (R[r_{of} \to addr_{of}])\,\mathbb{R} \qquad \mathbb{H}[addr_{of} \to Ref_{of}]\rangle}{\mathscr{P}\,\langle \mathbb{I}\rangle}$$

given $\quad \delta_{instance?}(Ref_{of},\ \textit{Ref-Type})$

$$\overset{\text{INSTANCE-OF}}{\underset{R,\,\mathbb{H}}{\hookrightarrow}} \frac{\mathscr{P}\,\langle (\texttt{instance-of}\ r_{to}\ r_{of}\ \textit{Ref-Type})\,\mathbb{I} \qquad (R[r_{of} \to addr_{of}])\,\mathbb{R} \qquad \mathbb{H}[addr_{of} \to Ref_{of}]\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := i_{bool}])\,\mathbb{R}\rangle}$$

where $\quad i_{bool} = \begin{cases} 1 & \text{if} \quad \delta_{instance?}(Ref_{of},\ \textit{Ref-Type}) \\ 0 & \text{otherwise} \end{cases}$

(i) Casting

Figure 7: Welterweight Dalvik reduction rules

These rules handle run-time type-checking.

- The CHECK-CAST rule checks if the object referenced through r_{of} is an instance of the given reference type *Ref-Type*, using the $\delta_{instance?}$ metafunction; if it isn't, the program enters a stuck state.
- The INSTANCE-OF rule does the same as CHECK-CASE, but returns an interger-encoded boolean reflecting the type-check through the destination register r_{to}.

ARRAY-LEN

$$\hookrightarrow_{R,\,\mathbb{H}} \frac{\mathscr{P}\,\langle\,(\texttt{array-len}\ r_{to}\ \textit{Ref-Type}_{arr})\ \mathbb{I} \quad (R[r_{arr} \to \textit{addr}_{arr}])\,\mathbb{R} \qquad \mathbb{H}[\textit{addr}_{arr} \to (\texttt{array}\ \_\ \textit{Size}\ \_)]\rangle}{\mathscr{P}\,\langle\,\mathbb{I} \qquad (R[r_{to} := \textit{Size}])\,\mathbb{R}\rangle}$$

AGET

$$\hookrightarrow_{R,\,\mathbb{H}} \frac{\mathscr{P}\,\langle\,(\texttt{aget}\ r_{to}\ r_{arr}\ r_{index})\ \mathbb{I} \quad (R[r_{arr} \to \textit{addr}_{arr},\ r_{index} \to i_{index}])\,\mathbb{R} \qquad \mathbb{H}[\textit{addr}_{arr} \to (\texttt{array}\ \textit{Type}_{elem}\ \textit{Size}\ \textit{Elems})]\rangle}{\mathscr{P}\,\langle\,\mathbb{I} \qquad (R[r_{to} := v])\,\mathbb{R}\rangle}$$

given $\quad i_{index} < \textit{Size}$

where $\quad v = \begin{cases} \textit{Elems}(i_{index}) & \text{if} \quad i_{index} \in \textit{Elems} \\ v_{default} & \text{otherwise} \end{cases}$

where $\quad v_{default} = \begin{cases} 0 & \text{if} \quad \textit{Type}_{elem} \subseteq \textit{Prim-Type} \\ \texttt{null} & \text{if} \quad \textit{Type}_{elem} \subseteq \textit{Ref-Type} \end{cases}$

APUT

$$\hookrightarrow_{\mathbb{R}} \frac{\mathscr{P}\,\langle\,(\texttt{aput}\ r_{from}\ r_{arr}\ r_{index})\ \mathbb{I} \quad (R[r_{from} \to v,\ r_{arr} \to \textit{addr}_{arr},\ r_{index} \to i_{index}])\,\mathbb{R} \qquad \mathbb{H}[\textit{addr}_{arr} \to a]\rangle}{\mathscr{P}\,\langle\,\mathbb{I} \qquad \mathbb{H}[\textit{addr}_{arr} := a_{new}]\rangle}$$

given $\quad i_{index} < \textit{Size}$

given $\quad v = \begin{cases} \textit{addr} & \text{if} \quad \textit{Type}_{elem} \subseteq \textit{Ref-Type} \ \wedge\ \delta_{instance?}(\mathbb{H}(\textit{addr}),\ \textit{Type}_{elem}) \\ \texttt{null} & \text{if} \quad \textit{Type}_{elem} \subseteq \textit{Ref-Type} \\ i & \text{if} \quad \textit{Type}_{elem} \subseteq \textit{Prim-Type} \end{cases}$

where $\quad (\texttt{array}\ \textit{Type}_{elem}\ \textit{Size}\ \textit{Elems}) = a$

where $\quad a_{new} = (\texttt{array}\ \textit{Type}_{elem}\ \textit{Size}\ \textit{Elems}[i_{index} \to v])$

(j) Array operations

Figure 7: Welterweight Dalvik reduction rules

These rules handle array operations.

- The ARRAY-LEN rule returns through the destination register r_{to} the size $\textit{Size}$ of the array referenced through r_{arr}.
- The AGET rule returns through the destination register r_{to} the element at the index given by the value in register r_{index} of the array referenced through r_{arr}. If the index is out of the bounds of the array's $\textit{Size}$, then the program enters a stuck state. If the index in question has not been assigned in the array, then a default value is used, according to the semantics of the DVM.

- The APUT rule assigns the index given by the value in register r_{index} of the array referenced through r_{arr} to the value in register r_{from}. If the index is out of the bounds of the array's $Size$, then the program enters a stuck state. If the value being put does not conform to the array's type given by $Type_{elem}$, then the program enters a stuck state.

$$\text{IGET}\quad \frac{\mathscr{P}\,\langle\,(\texttt{iget}\ r_{to}\ r_{obj}\ \mathit{Field\text{-}Spec})\ \mathbb{I} \quad (R[r_{obj} \to addr_{obj}])\,\mathbb{R} \quad \mathbb{H}[addr_{obj} \to (\texttt{obj}\ \mathit{FQ\text{-}Class\text{-}Spec}\ \mathit{IFields})]\rangle}{\mathscr{P}\,\langle\,\mathbb{I} \quad (R[r_{to} := v])\,\mathbb{R}\rangle} \quad \hookrightarrow_{\mathscr{C},\,R,\,\mathbb{H}}$$

where $v = \begin{cases} \mathit{IFields}(\mathit{Field\text{-}Spec}) & \text{if} \quad \mathit{Field\text{-}Spec} \in \mathit{IFields} \\ v_{default} & \text{otherwise} \end{cases}$

where $(\texttt{field}\ \mathit{Field\text{-}Spec}\ \_\ \_\ \_\ v_{default}) = \delta_{field}(\mathscr{C}(\mathit{FQ\text{-}Class\text{-}Spec}),\ \mathit{Field\text{-}Spec})$

$$\text{IPUT}\quad \frac{\mathscr{P}\,\langle\,(\texttt{iput}\ r_{from}\ r_{obj}\ \mathit{Field\text{-}Spec})\ \mathbb{I} \quad (R[r_{from} \to v,\, r_{obj} \to addr_{obj}])\,\mathbb{R} \quad \mathbb{H}[addr_{obj} \to o]\rangle}{\mathscr{P}\,\langle\,\mathbb{I} \quad \mathbb{H}[addr_{obj} := o_{new}]\rangle} \quad \hookrightarrow_{\mathscr{C},\,\mathbb{R}}$$

given $v = \begin{cases} addr & \text{if} \quad \mathit{Type}_{field} \subseteq \mathit{Ref\text{-}Type} \ \wedge\ \delta_{instance?}(\mathbb{H}(addr),\ \mathit{Type}_{field}) \\ \texttt{null} & \text{if} \quad \mathit{Type}_{field} \subseteq \mathit{Ref\text{-}Type} \\ i & \text{if} \quad \mathit{Type}_{field} \subseteq \mathit{Prim\text{-}Type} \end{cases}$

where $(\texttt{field}\ \mathit{Field\text{-}Spec}\ \_\ \_\ \mathit{Type}_{field}\ \_) = \delta_{field}(\mathscr{C}(\mathit{FQ\text{-}Class\text{-}Spec}),\ \mathit{Field\text{-}Spec})$

where $(\texttt{obj}\ \mathit{FQ\text{-}Class\text{-}Spec}\ \mathit{IFields}) = o$

where $o_{new} = (\texttt{obj}\ \mathit{FQ\text{-}Class\text{-}Spec}\ \mathit{IFields}[\mathit{Field\text{-}Spec} \to v])$

(k) Instance operations

Figure 7: Welterweight Dalvik reduction rules

These rules handle instance operations.

- The IGET rule returns through the destination register r_{to} the value that the object referenced through r_{obj} has for its field given by *Field-Spec*. If the object doesn't have a value for that field, then the default for that field is used, given its definition found using the metafunction δ_{field}.

- The IPUT rule assigns the field given by *Field-Spec* of the object referenced through r_{obj} to the value in register r_{from}. If the value being put does not conform to the field's type given by Type_{field}, then the program enters a stuck state.

$$\underset{R,\,\mathbb{H}}{\hookrightarrow} \quad \frac{\text{SGET}\quad \mathscr{P}\,\langle (\texttt{sget}\ r_{to}\ \textit{Field-Spec})\,\mathbb{I} \qquad R\,\mathbb{R} \qquad \mathbb{H}[\textit{FQ-Field-Spec} \to v]\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad (R[r_{to} := v])\,\mathbb{R}\rangle}$$

$$\underset{\mathscr{F},\,\mathbb{R}}{\hookrightarrow} \quad \frac{\text{SPUT}\quad \mathscr{P}\,\langle (\texttt{sput}\ r_{from}\ \textit{Field-Spec})\,\mathbb{I} \qquad (R[r_{from} \to v])\,\mathbb{R} \qquad \mathbb{H}\rangle}{\mathscr{P}\,\langle \mathbb{I} \qquad \mathbb{H}[\textit{FQ-Field-Spec} := v]\rangle}$$

$$\text{given} \quad v = \begin{cases} \textit{addr} & \text{if} \quad \textit{Type}_{field} \subseteq \textit{Ref-Type} \ \wedge\ \delta_{instance?}(\mathbb{H}(\textit{addr}),\ \textit{Type}_{field}) \\ \texttt{null} & \text{if} \quad \textit{Type}_{field} \subseteq \textit{Ref-Type} \\ i & \text{if} \quad \textit{Type}_{field} \subseteq \textit{Prim-Type} \end{cases}$$

$$\text{where} \quad (\texttt{field}\ \textit{Field-Spec}\ \_\ \_\ \textit{Type}_{field}\ \_) = \mathscr{F}(\textit{FQ-Field-Spec})$$

(l) Static operations

Figure 7: Welterweight Dalvik reduction rules

These rules handle static field operations.

- The SGET rule returns through the destination register r_{to} the value of the static field given by *FQ-Field-Spec*.
- The SPUT rule assigns the field given by *FQ-Field-Spec* to the value in register r_{from}. If the value being put does not conform to the field's type given by $\textit{Type}_{field}$, then the program enters a stuck state.

INVOKE-VIRTUAL

$$\frac{\mathscr{P}\,\langle\,(\texttt{invoke-virtual}\ \mathit{Method\text{-}Spec}\ r_{this}\ r_{args}\ \ldots)\,\mathbb{I}\quad (R[r_{this} \to addr_{this}])\,\mathbb{R}\qquad \mathbb{H}[addr_{this} \to o_{this}]\rangle}{\mathscr{P}\,\langle\,(\texttt{invoke}\ \mathit{Method}\ r_{r_{this}\ r_{args}\ldots}\ \ldots)\,\mathbb{I}\rangle}\ \mathscr{C},\mathbb{R},\mathbb{H}$$

given $\texttt{static} \notin [\mathit{Attr}_{method}\ \ldots]$
where $(\texttt{method}\ \_\ \_\ [\mathit{Attr}_{method}\ \ldots]\ \_\ \_\ \_\ \ldots) = \mathit{Method}$
where $\mathit{Method} = \delta_{method}(\mathscr{C}(\mathit{FQ\text{-}Class\text{-}Spec}),\ \mathit{Method\text{-}Spec})$
where $(\texttt{obj}\ \mathit{FQ\text{-}Class\text{-}Spec}\ \_) = o_{this}$

INVOKE-SUPER

$$\frac{\mathscr{P}\,\langle\,(\texttt{invoke-super}\ \mathit{Method\text{-}Spec}\ r_{this}\ r_{args}\ \ldots)\,\mathbb{I}\quad (R[r_{this} \to addr_{this}])\,\mathbb{R}\qquad \mathbb{H}[addr_{this} \to o_{this}]\rangle}{\mathscr{P}\,\langle\,(\texttt{invoke}\ \mathit{Method}\ r_{r_{this}\ r_{args}\ldots}\ \ldots)\,\mathbb{I}\rangle}\ \mathscr{C},\mathbb{R},\mathbb{H}$$

given $\texttt{static} \notin [\mathit{Attr}_{method}\ \ldots]$
where $(\texttt{method}\ \_\ \_\ [\mathit{Attr}_{method}\ \ldots]\ \_\ \_\ \_\ \ldots) = \mathit{Method}$
where $\mathit{Method} = \delta_{method}(\mathscr{C}(\mathit{FQ\text{-}Super}),\ \mathit{Method\text{-}Spec})$
where $(\texttt{class}\ \_\ \_\ \_\ \mathit{Super}\ \_\ \_) = \mathscr{C}(\mathit{FQ\text{-}Class\text{-}Spec})$
where $(\texttt{obj}\ \mathit{FQ\text{-}Class\text{-}Spec}\ \_) = o_{this}$

INVOKE-STATIC

$$\frac{\mathscr{P}\,\langle\,(\texttt{invoke-static}\ \mathit{Method\text{-}Spec}\ r_{args}\ \ldots)\,\mathbb{I}\rangle}{\mathscr{P}\,\langle\,(\texttt{invoke}\ \mathit{Method}\ r_{r_{this}\ r_{args}\ldots}\ \ldots)\,\mathbb{I}\rangle}\ \mathscr{M}$$

given $\texttt{static} \in [\mathit{Attr}_{method}\ \ldots]$
where $(\texttt{method}\ \_\ \_\ [\mathit{Attr}_{method}\ \ldots]\ \_\ \_\ \_\ \ldots) = \mathit{Method}$
where $\mathit{Method} = \mathscr{M}(\mathit{FQ\text{-}Method\text{-}Spec})$

INVOKE

$$\frac{\mathscr{P}\,\langle\,(\texttt{invoke}\ \mathit{Method}\ r_{args}\ \ldots)\,\mathbb{I}\qquad (R[r_{args}\ \ldots \to v_{args}\ \ldots])\,\mathbb{R}\qquad \mathbb{S}\qquad \mathbb{N}\qquad \mathbb{E}\qquad \mathbb{M}\rangle}{\mathscr{P}\,\langle\,\mathbb{I} : [\mathit{Stmt}_{body}\ \ldots]\qquad ()\,R\,\mathbb{R}\qquad (v_{args}\ \ldots)\,\mathbb{S}\qquad \mathbb{I}\,\mathbb{N}\qquad []\,\mathbb{E}\qquad \mathit{Method}\,\mathbb{M}\rangle}\ R$$

given $v = \begin{cases} addr & \text{if}\quad \mathit{Type}_{param} \subseteq \mathit{Ref\text{-}Type}\ \wedge\ \delta_{instance?}(\mathbb{H}(addr),\ \mathit{Type}_{param}) \\ \texttt{null} & \text{if}\quad \mathit{Type}_{param} \subseteq \mathit{Ref\text{-}Type} \\ i & \text{if}\quad \mathit{Type}_{param} \subseteq \mathit{Prim\text{-}Type} \end{cases}$
for $v \in [v_{args}\ \ldots],\ \mathit{Type}_{param} \in [\mathit{Type}_{params}\ \ldots]$
where $(\texttt{method}\ \_\ \_\ \_\ [\mathit{Type}_{params}\ \ldots]\ \_\ \mathit{Stmt}_{body}\ \ldots) = \mathit{Method}$

(m) Method invocation

Figure 7: Welterweight Dalvik reduction rules

These rules handle method invocation.

- The INVOKE-VIRTUAL rule transitions to an `invoke`, using the instance method ***Method*** identified by the metafunction δ_{method}, using the class ***Class*** given by looking up in the class registry $\mathscr{C}$ the ***FQ-Class-Spec*** of the object referenced through r_{this}. If ***Method*** is declared to be static, then the program enters a stuck state.

- The INVOKE-SUPER rule behaves as in INVOKE-VIRTUAL, but uses o_{this}'s superclass for the method lookup.

- The INVOKE-STATIC rule also transitions to an `invoke`, using the static method ***Method*** given by looking up in the method registry $\mathscr{M}$ the given ***FQ-Method-Spec***. If ***Method*** is not declared to be static, then the program enters a stuck state.

- The INVOKE rule is the actual workhorse of invocation. It pushes new frames onto all method-scoped stacks, in particular pushing the values in the argument registers $r_{args} \ldots$ onto the $\mathbb{S}$, the remaining instruction stream $\mathbb{I}$ onto the $\mathbb{N}$, and the invoked method ***Method*** onto the $\mathbb{M}$. It jumps to ***Method***'s body by setting the instruction stream. If any of the argument values do not conform to ***Method***'s corresponding formal parameter type given by $\mathit{Type}_{params} \ldots$, then the program enters a stuck state.

Chapter III

OOCFA2

This chapter describes the OOCFA2 algorithm and the structures behind it in detail. In section 3.1 we discuss the structuring and implementation of the value space for both the conceptual version and implementation of the OOCFA2 algorithm. In section 3.2 we discuss the algorithm itself, present it in pseudocode, and provide miscellaneous notes and caveats and limitations regarding the algorithm. In section 3.3 we discuss what we consider to be the most noteworthy extensions that our algorithm provides on top of more traditional flow analyses and abstract interpretations. Finally, in section 3.4 we provide an empirical evaluation of the algorithm's implementation and discuss the results.

As a very general description of the algorithm, consider once again that it is an abstract interpretation of the DVM ROP. Thus, we work with a model of Java's run-time, including an abstract heap, stack, registers, static fields, and all supporting infrastructure such as the class registry. During the interpretation, we keep track of the set of possible concrete values an abstract value can take. We could also have no idea what the possible concrete values of an abstract value could be (e.g., if it is input), and so we consider it unknown (though we do have some idea of its type). Using these abstract values, we mimic the ROP instructions on our abstracted state. If we come to a point where, given multiple or unknown concrete values, there are multiple possible control flows, we explore them all, using a different set of abstract states for each. We avoid exploring the same abstract state twice by keeping track of those we've visited and remembering their outcomes; this is termed **summarization** and is crucial to the tractability of our algorithm. We also include some heuristics (section 3.3) to prevent infinite (or too much) looping in the abstract interpretation, even if the abstract states are distinct.

We have implemented our algorithm in the Scala programming language [28]. This is a language which is compatible with Java and runs on the JVM. As such, our implementation is idiosyncratically designed to "reflect" or "lift" the Java type system into itself, as seen from Scala. We explain this design in subsection 3.1.1.

We now give a brief overview of type theory and related terminology which we use or introduce. In a type-theoretic sense, *type*s categorize values and *kind*s are types of types. In order to generalize these two concepts, we introduce the notion of a **variety**: types are

varieties of values and kinds are varieties of types. A **dependent variety** is a variety which depends upon that over which it varies, e.g., a dependent type depends on values. A **singleton variety** is a dependent variety which denotes a set consisting of a single thing over which it varies, often aliased with itself, e.g., the integer 0 can be thought of as typed by the singleton type of 0 (itself kinded by, e.g., *Nat*). We use Guy Steele's recently-coined terminology of *ilk* [3] as the definition of the "run-time type" of a value. In his phrasing, "expressions have types; values have ilks". Finally, Java has two instantiable classes of `Throwable` (which denotes values which can participate in exception-handling): `Error` and `Exception`. We collectively refer to values of either of these classes as **exceptionals**.

3.1 Value Space

At a high-level view, **environment**s map **variable**s (*vars*) to **abstract value**s ($\widehat{val}s$). In turn, $\widehat{val}$s represent sets of possible concrete values (*vals*) or an unknown value ($\top$).

A **trace environment** is used for intra-function, inter-basic-block variables, which are **register variable**s.

A **static environment** is used for inter-function, *static field*s, which in Java semantics are attached to a particular class and not its instances. These fields are denoted by **static variable**s.

An **instance environment** is used for the association between the instance-field slots of the ilk of an (*Object* -derived) instance, denoted by **field variable**s, and the values for that instance's fields. In this way, every "object" is an environment in and of itself, mirroring the semantics of object-orientation.

Finally, a **heap environment** is used for tracking the values on the abstract heap according to their abstract heap address, denoted by **heap variable**s.

Each concrete value is annotated upon creation with the abstract values it depends upon. For example, when performing an addition, the resulting value is noted as dependent upon the operands to the addition instruction. This is important for proper handling of looping/recursion (subsection 3.3.1), besides being useful for a variety of analyses layered atop OOCFA2, such as information-flow analysis left for future work (subsubsection 6.1.3.1).

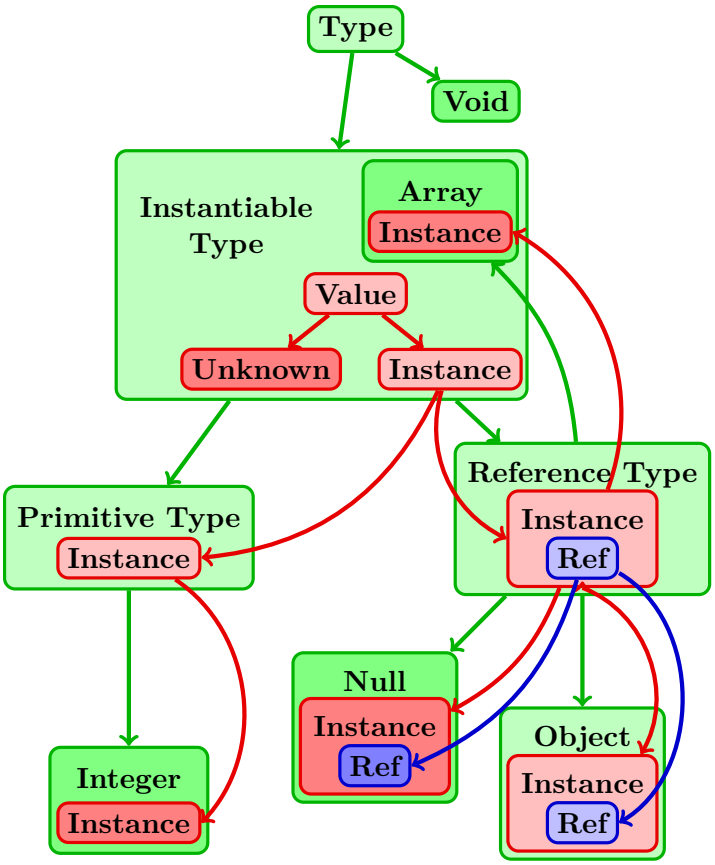

Figure 8: The implementation of OOCFA2's value-space, with respect to Java types

3.1.1 Implementation

The implementation of the value space is modeled after the typing relations of the Java type system, which Dalvik inherits. In Figure 8, we give a diagram of a highlighted portion of the value space (incomplete and eliding many details) to help visualize the following discussion.

In the Java type system, ilks are given by first-class entities called *classes*. Notably, this includes even the primitive, non-referential types, such as integers; they too have ilks defined by classes, though the association between the two is only emulated by the system; they have no direct tie to their ilk's "class". We model ilks as instances (in the Java sense) of Java (Scala) classes, which we use to model kinds. In turn, *values* of a particular class ***C*** are modeled as instances of ***C***'s particular **Value** inner-class; more specifically as instances of its **Unknown** inner-class should the values be unknown or as instances of its **Instance** inner-class if the value is known. In this manner, we "lower" the Java type system one

"level", such that we can in a more intuitive manner emulate the type-driven operations of the language.

In Figure 8, arrows denote object-oriented inheritance, encapsulation denotes inner-classing, and blocks of a darker color denote singleton classes. Green blocks denote "kind-classes" (or types as well, if they are singleton kinds), red blocks denote "value-classes", and blue blocks denote "reference-classes".

At the top-level, there is **Type**, encompassing the entire system. There are quite a few sub-kinds of **Type** which exist in the system, but of particular note is the **Instantiable** sub-kind. It denotes the kind of all types which represent values which can exist at run-time (e.g., not the **Void** type). It first defines the inner-classes of **Value**, **Unknown**, **Instance**, and **Array**. As Scala does not support "virtual inner classes", sub-kinds are contractually (forced by the type system) obligated to have their own versions of those inner-classes extend their super-kind's versions, should they "override" (redefine, really) them.

Though the other inner-classes were previously discussed, **Array** was not; it is actually a singleton kind. It cannot be extended (thus a singleton kind effectively doubling as a type) and is, intuitively, the type of arrays of its enclosing type. It is dependent in that it depends on the specific type of its parent kind (the instance of its parent class). In normal circumstances, one would express this by making **Array** a type which is parameterized over the type of its elements, but as types are first-class instances, this would operate on the wrong level (indeed, requiring dependent types). Instead, we emulate a dependent kind in the Java-standard way: using an inner-class.

Of note, **Array** is actually a sub-kind of **Reference Type**. The latter denotes the kind of all types which are reference types in the Java system. Notably, this includes not only Java's "top" *Object* type and all of its classes, but also **Array** as mentioned and the special **Null** singleton kind. The latter exists because the Dalvik system provides a type-descriptor for values which are known to be null. Reference types' **Instance**s themselves include one more layer of inner-classing: the **Ref** type. These represent the type of references to the particular concrete value (instance of **Instance**) in question, again emulating a dependent type with an inner-class.

Finally, **Object** is the singleton kind representing the *Object* class at the top of the traditional Java class hierarchy. Its **Instance** type provides the analysis' modeling of instance fields, their environments, their inheritance and overriding behavior, etc.

A class ***C*** which has been seen during analysis (either because it was mentioned in the type of an object or method, or because ***C*** itself is being analyzed) is modeled by creating an instance of the **Object** kind, and thus ***C*** is a value representing a type. Note that as all of the primitive types are built into the analysis, the only possible "novel" types that can be encountered during analysis must be reference types; and as array-types are constructed from their element types and the pseudo-type **Null** is inextensible, this leaves **Object** as the only possible super-kind/type for the newly-constructed type in question.

Using reflection-hackery and dependency-injection on the JVM, it would also be possible to dynamically create a Java class to represent ***C*** and make it extend the **Object** type. This is actually what is done (in a hard-coded manner instead of a reflective one) for the analysis' built-in, common types (e.g., *String* and *Exception*). However, it would be of little practical benefit given that the machinery works as-is and that, as Scala does not (and may never) support "virtual inner classes", the inner **Value**-derived classes would have to be manually extended from their super-kind versions, and thus the burden and level of hackery is too great to validate this scheme.

3.2 Algorithm

A high-level overview of the implementation's algorithm is as follows:

1. We analyze each accessible method (including "main") – that is, public or protected in a non-final class – with an unknown static environment (or the initial static environment if analyzing "main").

2. To analyze a method, we abstractly interpret over its basic blocks.

3. To interpret a basic block, we evaluate each of its instructions in sequence, accumulating changes to the static environment, heap, register-file, etc.

4. If we encounter a method call, then we set up to call it (e.g., pushing parameters

on the stack) and jump to analyzing it in this new context, assuming it hasn't been summarized.

5. If the basic block ends in an unconditional branch, we simply begin interpreting the following basic block.

6. If the basic block ends with a return, then we update the summary of the method we have been interpreting and return back to the calling context in our analysis.

7. If the basic block ends with a conditional, we check to see if we statically know the outcome of the conditional operation and if so only interpret the basic block along the arm of the branches (plural in the case of a switch statement) which are possible. Either way, we propagate conditional-dependent information along the corresponding arms.

The pseudocode in Figure 9 gives a more detailed view of the implementation's algorithm. Note that we use a few conventions in these figures:

- We use the ⦇⊸⦈ notation to denote a multiset

- Some function calls are double-underlined. This is to indicate that, in actuality, this function is defined in the scope of the underlined call; it is effectively "inlined". Thus, e.g., all variables visible in such a scope are also visible in the function's definition, as long as they are not shadowed by, e.g., the function's parameters.

- Some function calls are single-underlined. This is a visual reminder that the function being called is defined within this scope's parent function.

```
1  options ← options provided by user
2  accessible-methods ← all accessible methods in compilation unit
3  init-static-env, init-heap-env ← initial environments, e.g., after class initialization
4  summaries : {METHOD ↦ FINDEX ↦ FSUMMARY}
               ← ∅
5  f-effects : {FTRACE ↦ FEFFECT}
              ← ∅
6  for method ← accessible-methods
7      f-trace ← ∅
8      f-phases : {METHOD ↦ FTRACE ↦ PHASE}
                  ← ∅
9      unknown-params ← array of unknown instances of each parameter's declared type
10     TRACE-METHOD(method, f-trace, f-phases,
                    init-static-env, init-heap-env, unknown-params)
```

(a) Pseudocode for the top level of OOCFA2

Figure 9: OOCFA2 pseudocode

```
TRACE-METHOD(m, f-trace, f-phases, static-env, heap-env, params)
 1  append m to f-trace
 2  header-trace ← f-trace relative to m
 3  phase ← judge phase from f-phases using header-trace
 4  match phase
 5  case NONE ∨ INIT ⇒ do nothing
 6  case STEP ⇒
 7       insert f-trace ↦ (static-env, heap-env) into f-effects[m]
 8  case CLEANUP ⇒
 9       last-effect ← f-effects[m][header-trace]
10       induce unknowns in (static-env, heap-env) relative to last-effect
11       insert f-trace ↦ (static-env, heap-env)
                 into f-effects[m]
    // DONE case is impossible
12  f-phases[header-trace] ← ADVANCE-PHASE(phase, header-trace, options.recursion-fuel)
13  f-index ← (static-env, heap-env, params)
14  if ∃summary : summaries[m][f-index]
15       return summary
16  else
17       bb-effects : {BBTRACE ↦ BBEFFECT}
                    ← ∅
18       bb-phases : {BB ↦ BBTRACE ↦ PHASE}
                    ← ∅
19       bb-trace ← ∅
20       eval-state ← ∅
21       uncaught-exns ← ∅
22       trace-env ← ∅
23       TRACE-BB(m.prologue, bb-trace,
                    eval-state, uncaught-exns, static-env, heap-env, trace-env)
24       return summaries[m][f-index]
```

(b) Pseudocode for the TRACE-METHOD function, defined within the top-level

Figure 9: OOCFA2 pseudocode

```
TRACE-BB(bb, bb-trace, eval-state, uncaught-exns, static-env, heap-env, trace-env)
 1  append bb to bb-trace
    // Note that result here is an alias for the operation's sink register in trace-env
 2  for ins ∈ bb.instructions match ins.operation
 3  case NOP ⇒ do nothing
 4  case op : EVALUABLE ⇒
 5      result ← evaluate according to op.opcode using op.operands
 6  case op : MOVE ⇒
 7      result ← take from op.operands or eval-state, depending upon op.opcode
 8  case op : GET-FIELD ⇒
 9      result ← specified field from trace-env[op.operands[0]]
10  case op : GET-STATIC ⇒
11      result ← specified field from static-env
12  case op : PUT-FIELD ⇒
13      set specified field in trace-env[op.operands[1]] using trace-env[op.operands[0]]
14      update trace-env
15  case op : PUT-STATIC ⇒
16      set specified field in static-env using trace-env[op.operands[0]]
17  case op : A-GET ⇒
18      result ← value in index trace-env[op.operands[1]]
                          of array trace-env[op.operands[0]]
19  case op : A-PUT ⇒
20      put trace-env[op.operands[2]] in index trace-env[op.operands[1]]
                                          of array trace-env[op.operands[0]]
21      update trace-env
22  case op : MONITOR ⇒
23      set monitored status of op.operands[0]
24      update trace-env
25  case op : NEW-INSTANCE ⇒
26      result ← create a new instance of the specified class
27  case op : (NEW-ARRAY ∨ FILLED-NEW-ARRAY) ⇒
28      result ← create a new array (possibly filled) with elements of the specified class
29  case op : CHECK-CAST ⇒
30      obj ← trace-env[op.operands[0]]
31      if obj might not satisfy the specified type
32          add the appropriate CLASSCASTEXCEPTION to uncaught-exns
33          if obj must satisfy the specified type
34              result ← obj operating at the specified type
35          elseif obj might satisfy the specified type
36              result ← an unknown instance of the specified type
37      elseif obj does not satisfy the specified type
38          FOLLOW-PATHS(CALCULATE-CATCHERS(op))
39          FINALIZE-TRACE-BB
40  case op : CONST ⇒
41      result ← extracted constant from op
42  case op : ARRAY-LENGTH ⇒
43      result ← length of array given by op.operands[0]
44  case op : CALL ⇒ HANDLE-CALL
45  case op : BRANCH ⇒ HANDLE-BRANCH
```

(c) Pseudocode for the TRACE-BB procedure, defined within TRACE-METHOD

Figure 9: OOCFA2 pseudocode

```
1  params ← extracted parameters from op
2  if op.opcode ≠ INVOKE-STATIC
3       prepend "this" object (extracted from op) to params
4  possible-targets ← method resolved from op according to its opcode
                                          and "this" object's ilk, if applicable
5  for target ∈ possible-targets
6       if target is unknown
7            conservative-summary ← unknown value of the return type and
                                          any possible exception thrown from the signature
8            EVAL-SUMMARY(conservative-summary)
9       else
10           next-header ← f-trace relative to target
11           next-phase ← judge phase from f-phases using next-header
12           if next-phase = DONE
13                handle target as if it were unknown
14           else EVAL-SUMMARY(TRACE-METHOD(target, f-trace, f-phases,
                                          static-env, heap-env, params))
```

(d) Pseudocode for the HANDLE-CALL procedure, separated from TRACE-BB

Figure 9: OOCFA2 pseudocode

```
EVAL-SUMMARY(summary)
1  update ret-val in eval-state using summary
2  update exn-val in eval-state using summary
3  update uncaught-exns using summary
```

(e) Pseudocode for the EVAL-SUMMARY procedure, defined within HANDLE-CALL

Figure 9: OOCFA2 pseudocode

```
1  if bb.successors.size = 1
       // Special hotpath for unambiguous succession; also subsumes GOTO
2      if op may end
3          HANDLE-END
4      TRACE-BB(bb.successors.head, bb-trace,
                eval-state, uncaught-exns, static-env, heap-env, trace-env)
5      return
6  elseif bb.successors.size = 0
7      HANDLE-END
8      FINALIZE-TRACE-BB
9  else
10 header-trace ← bb-trace relative to bb
11 phase ← judge phase from bb-phases using header-trace
12 match phase
13 case NONE ∨ INIT ⇒ do nothing
14 case STEP ⇒
15     insert bb-trace ↦ (static-env, heap-env, trace-env, uncaught-exns)
           into bb-effects[bb]
16 case CLEANUP ⇒
17     last-effect ← bb-effects[bb][header-trace]
18     induce unknowns in (static-env, heap-env, trace-env, uncaught-exns)
                       relative to last-effect
19     insert bb-trace ↦ (static-env, heap-env, trace-env, uncaught-exns)
           into bb-effects[bb]
20 case DONE ⇒ return // Done cleaning up loop
21 bb-phases[trace-header] ← ADVANCE-PHASE(phase, header-trace, options.looping-fuel)
22 reach ← CALCULATE-CATCHERS(op) ∪ CALCULATE-REACHABLE-SUCCESSORS(op)
23 if reach ≠ ∅
24     FOLLOW-PATHS(reach)
25 else FINALIZE-TRACE-BB
```

(f) Pseudocode for the HANDLE-BRANCH procedure, separated from TRACE-BB

Figure 9: OOCFA2 pseudocode

```
1  match op.opcode
2  case RETURN ⇒
3      if op.sources.size = 0
4          value ← VOID
5      else
6          value ← trace-env[op.operands[0]]
7          insert bb-trace ↦ value into f-summaries[m][f-index].ret-vals
8  case THROW ⇒
9      add op.operands[0] to uncaught-exns
```

(g) Pseudocode for the HANDLE-END procedure, defined within HANDLE-BRANCH

Figure 9: OOCFA2 pseudocode

```
FOLLOW-PATHS(paths)
1  for (successor, bb-patch) ∈ paths
2      locally update (eval-state, uncaught-exns, static-env, heap-env, trace-env)
                  using bb-patch
3      TRACE-BB(successor, bb-trace,
                eval-state', uncaught-exns', static-env', heap-env', trace-env')
```

(h) Pseudocode for the FOLLOW-PATHS procedure

Figure 9: OOCFA2 pseudocode

```
1  insert (bb-trace ↦ uncaught-exns into f-summaries[m][f-index].uncaught-exns
2  return
   // Note that since this procedure is inlined, this returns from the enclosing function
```

(i) Pseudocode for the FINALIZE-TRACE-BB procedure

Figure 9: OOCFA2 pseudocode

```
CALCULATE-CATCHERS(op)
 1  catchers : ⟅BB ⊸ BRANCHPATCH⟆
          ← ∅
 2  for t ∈ op.catch-types
 3      handlers ← bb.handlers-for(t)
 4      for exn ∈ uncaught-exns
 5          tri ← t > exn.type
 6          if tri ≠ ⊥
 7              for handler ∈ handlers
 8                  branch-patch ← ⟨eval-state.exn-val ← exn
                                   remove exn from uncaught-exns ⟩
 9                  insert handler ↦ branch-patch into catchers
10          if tri = ⊤
11              remove exn from uncaught-exns
12 return catchers
```

(j) Pseudocode for the CALCULATE-CATCHERS function

Figure 9: OOCFA2 pseudocode

```
CALCULATE-REACHABLE-SUCCESORS(op)
 1  reach : ⦇BB ⊸ BranchPatch⦈
          ← ∅
 2  match op
 3  case op : If ⇒
 4       tri ← evaluation of op's conditional with its operands
 5       if tri ≅ ⊤
 6            insert bb.successors[0] ↦ ⟨ conditional-/path-specific knowledge ⟩ into reach
 7       if tri ≅ ⊥
 8            insert bb.successors[1] ↦ ⟨ conditional-/path-specific knowledge ⟩ into reach
 9  case op : Switch ⇒
10       tris ← { evaluate op's conditional with its operand and entry | entry ∈ op}
11       if ∃! tri ∈ tris, successor ∈ bb.successors | tri = ⊤, tri ⊨ successor
12            insert successor ↦ ⟨ conditional-/path-specific knowledge ⟩ into reach
13       else for tri ∈ tris | successor ∈ bb.successors
14            if tri ≅ ⊤
15                 insert successor ↦ ⟨ conditional-/path-specific knowledge ⟩ into reach
16  return reach
```

(k) Pseudocode for the CALCULATE-REACHABLE-SUCCESSORS function

Figure 9: OOCFA2 pseudocode

3.2.1 Miscellany

As the implementation runs on the JVM, we take advantage of this fact and use the JVM's reflection abilities to "lift" certain pure methods into the analysis-proper, given that they operate on known (non-bottom) abstract values. For example, we use this ability to append strings together and perform certain static string methods; this allows us to perfectly accurately analyze common string behavior, whereas we wouldn't be able to do so were we to have to treat the Java API as a black-box. In particular, this string example is very useful with respect to Android because so many of its API-provided facilities rely on string parameters for, e.g., identification. One example of this is how one identifies intents by using their string name.

3.2.2 Caveats & Limitations

In the implementation, we have made a few – what we believe to be – relatively benign simplifying assumptions, besides the fact that the analysis is subject to some inherent limitations.

3.2.2.1 Unchecked exceptionals

The analysis assumes that **unchecked exceptionals** (*Error*s or *RuntimeException*s) would only be caught inside of the method in which they could occur. Therefore, if they aren't caught, then they would never be and the program would terminate anyway, thus there is no harm in erasing them should they not be caught. Furthermore, Sun (now Oracle) has—since Java's inception—explicitly recommended against catching unchecked exceptionals in application code [21], regarding it not only as bad practice, but as leading to potentially undefined behavior.

3.2.2.2 Multi-threading

The analysis does not handle multi-threading; it is outside the scope of this work. Our analysis assumes that during our abstract interpretation, the only way that variables can be changed is via the interpretation's single abstract thread of control. It would be bad form and antithetical to the "Android way" [11] to mutate shared global state between two

threads of control anyway, not to mention that this is a difficult and ongoing problem in the field of abstract analysis.

3.2.2.3 Whole-application view

The analysis assumes that it is given a view of an entire Android application in this compilation unit. This means that everything it is analyzing will be executed in its own singular VM at run-time, and thus the analysis doesn't have to worry about certain kinds of interference, such as static variables being assigned outside its purview. Therefore, it can reasonably guarantee the points by which the world interfaces with the analyzed code.

3.2.2.4 Reflection

Though noted in official Google documentation as an unsupported means of application programming, Java-provided reflection facilities are used by certain applications. Legitimate applications often make use of reflection to, e.g., deserialize JSON and XML, invoke private or otherwise hidden Android API methods, and abstract over API classes whose names changed between versions. [9] Of course, reflection is a common means of evading static analysis and is especially used in many malicious and black-market applications. The algorithm does not currently deal with reflection, but it should in principle not be too difficult to do so (subsubsection 6.1.2.1).

3.2.2.5 Java Native Interface

As the analysis operates on ROP, it obviously cannot analyze past the *JNI* ("Java Native Interface")—the boundary between the JVM/DVM and native code. Though ostensibly we could extend the analysis to analyze native code, this would be a tremendous and difficult undertaking, given the much more "open" semantics and environment surrounding native code.

3.2.2.6 Android-ignorance

The analysis does not currently encode any heuristics or knowledge of the Android environment. It only reasons over the DVM (in the form of ROP) and its semantics. Most Android development actually only nominally (in a sense) uses the Java language

and APIs. Instead, development is very Android-specific, using all sorts of Android-specific tools, frameworks, infrastructure, and metadata. As a result, Android applications typically have the following list of attributes:

- They are very "horizontal" in the sense that they do not create large amounts of deep structure for their own use, but rather provide a broad set of relatively shallow components which hook into the broad Android APIs and system appropriately.
- Following from the above point, they are very callback-oriented, in that the manner in which and points by which they hook into the Android APIs tend to be callbacks, event handlers, or asynchronous updates.
- Following from the above point, they are "implicitly" multi-threaded. Each thread is a separate, concurrent task which often behaves like an "actor" in the actor-model sense; it communicates with other actors in a structured manner and does far more computation on its own than involving communication.

As one example of a useful Android-specific heuristic, currently the implementation of OOCFA2 uses a very simple heuristic for deciding which methods are externally accessible and thus possible starting points for the analysis. It chooses all accessible methods according to the definition of the standard Java ecosystem: public methods or protected methods whose classes are non-final. This is a gross over-approximation of the Android system. Android applications do not directly link and live entirely in their own VM; thus these "accessible" methods cannot actually be accessed by external code. Rather, the starting points for an Android application are the methods which the APIs will use as callbacks.

Reasoning over the Android system—in this manner and others—is left for future work (subsection 6.1.2).

3.3 Flow-analysis Extensions

The following sections describe extensions to standard flow-analysis practices which are novel to the best of our knowledge.

3.3.1 Loop and recursion exploration

In order to guard against possibly infinite or undesirably large looping or recursion (hereafter interchangeably referred to as "looping"), we introduce a two new concepts: **header traces** and **loop phasing**.

A header trace is, relative to some current basic block *bb*, the sub-range of the current trace from the last time *bb* was encountered until the current point in the trace. It is used to identify looping by considering the header trace to be an iteration of a loop.

Each time the algorithm encounters an iteration of a loop – detected by passing though a previously-passed basic-block – the loop's phase (more accurately the header trace's phase) is advanced in the abstract interpretation. There are five phases in this scheme:

- *none*: The loop has not been entered yet.
- *init*: The loop has just been entered.
- *step*: The loop has iterated for a certain number of times. This is configurable akin (but *not equivalent*) to the traditional notion of "fuel "; the default is zero units of fuel, which equates to the step phase only consisting of a single iteration.
- *cleanup*: After all iterations in the step phase, cleanup is performed as discussed below.
- *done*: The analysis is done exploring this loop and thus should not continue to do so along this path of abstract interpretation.

Note that the notion of **fuel** used in OOCFA2 in the step phase differs importantly from the more traditional notion. Fuel in normal abstract-interpretation parlance describes the number of "steps" allowed to be taken in a potentially infinite looping or recursive interpretation before the analysis "gives up" and falls back to a far less precise (but still sound) result before moving on. In a similar spirit, OOCFA2's "fuel" reflects the number of *step phases* allowed; it basically reflects the number of times we've seen the particular loop (head trace) we're phasing over. However, it is important to note that an increase in fuel in OOCFA2 could easily lead to an exponential increase in the exploration of the analysis. This is due to the fact that each possible nondeterministic path abstractly interpreted past

the loop will be fully explored (and paths resulting from those paths, ad infinitum) until each path individually reaches the *done* phase of the loop in question.

The real trick to the precision of this approach (while preserving soundness) versus the typical fuel-driven approach involves the cleanup phase. When it is entered, the analysis examines the current register-file and heap and compares it to the last environment remembered for the particular header trace in question. For each abstract value which was created after two or more iterations of the loop, it is judged that the value is loop-varying and thus made $\top$ (it is **dropped**). Furthermore, every abstract value which depends in some way upon a dropped value is itself also dropped. In this manner, the algorithm ensures that any abstract values which varied with the loop are noted as unknown, because it cannot know how many times the loop could have executed, or even if that number of times was finite. The cleanup phase then proceeds, during which the unknown values are propagated through one iteration of the loop, in order to preserve soundness. Once the analysis has reached another iteration of the loop, it enters the "done" phase and ceases iterating.

3.3.2 Semantic aliasing

Semantic aliasing is a solution to the problem of undesirably disjoint environments. Consider the example given in Figure 10. In it, we have four basic blocks, $\boldsymbol{A}$ through $\boldsymbol{D}$. There is the allocation of some object o in $\boldsymbol{D}$ and some number of unnamed allocations in A. When abstractly interpreting this structure, we may follow the path $\boldsymbol{A} \rightarrow \boldsymbol{B} \rightarrow \boldsymbol{D}$ or the path $\boldsymbol{A} \rightarrow \boldsymbol{C} \rightarrow \boldsymbol{D}$. Now, let us assume (without loss of generality) that we have implemented abstract heap addressing and allocation via, e.g., a global counter, such that each allocation is guaranteed a different address throughout the entire abstract interpretation. If we were to follow the $\boldsymbol{A} \rightarrow \boldsymbol{B}$ path, o will be allocated some address a. If instead we follow the $\boldsymbol{A} \rightarrow \boldsymbol{C}$ path, o will be allocated some other address a' due to the intervening allocations in $\boldsymbol{C}$.

The problem lies in the fact that the user of the analysis – let us assume also the writer of the code being analyzed – clearly expects o to be a single, "traceable" object given the determinstic execution of the program. However, the analysis is inherently a

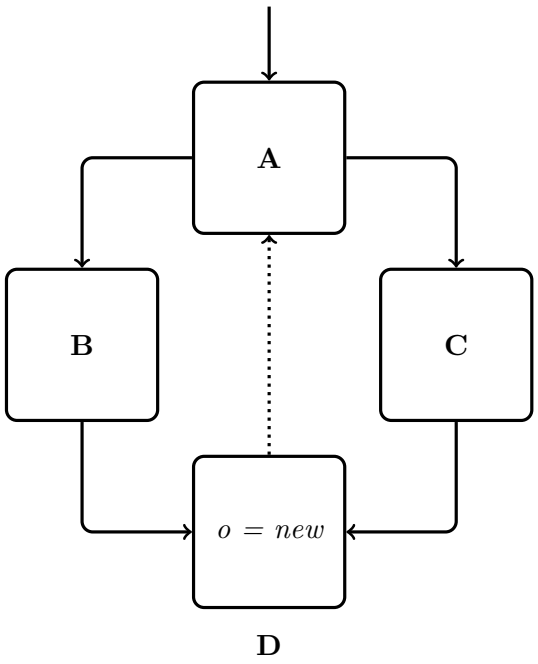

Figure 10: Semantic-aliasing basic-block example

*non*deterministic abstraction, exploring all *possible* paths, yielding different abstract heap addresses for o depending upon the path taken. Thus, if the user were to query the results of the analysis with, e.g., "what is are the possible outcomes of this function" (something common in, e.g., information flow analysis [subsubsection 6.1.3.1]), the union of the possible environments coming out of the function would include a and a' pointing to two versions of o, whereas the desired result would be a single a pointing to an *abstract value* denoting the two possibilities of o. (Better yet, the field environments of the two versions should be joined such that we see a single o with, e.g., an abstract value of two possibilities for some differing field f.) Thus, the algorithm should *induce* the *semantic aliasing* of the two possible os to a single heap address.

Note that the situation gets more complicated in the presence of loops (structured or otherwise), such as the hypothetical loop between D and A. In this case, the user does *not* expect o to represent a single conceptual "run-time object", but rather understands that, at run-time, the allocation instruction will allocate a new o for each time it is executed. Thus, semantic aliasing in this case is *undesirable* because it would impose aliasing on objects

which are *deterministically distinct* at run-time.

Our solution is that we address allocations to the abstract heap via a tuple of the instruction which induced the allocation and the trace of basic-blocks taken to get there. In this way, we have the information we require to properly evaluate – post-hoc – whether this is a semantic alias with respect to another allocation involving the same instruction, but along a *different* path. This is left to happen post-hoc and on-demand for the following reasons:

- The analysis is built in such a way that it does not use a worklist (at least not in the sense of CFA2 [25]), and thus abstractly interprets using the analysis' runtime stack. (This was inspired by jsCFA [24].) Because of this, the analysis does not join environments with other abstract traces of execution, given that, short of the set of function summaries, there *is no* cumulative record of environments to join.

- Future work involving loop-aware semantic aliasing (subsubsection 6.1.1.1) allows for different, dynamic, non-uniform notions of semantic aliasing and thus joining the environments immediately according to the notions described here would be inappropriate.

3.4 Empirical Evaluation

In this section, we give a preliminary evaluation of the accuracy of the OOCFA2 implementation. We also provide some performance data, though this is not particularly enlightening other than giving some reassurance that the analysis is not particularly "heavy".

3.4.1 Caveats

There are two important caveats to keep in mind when reviewing this data. First of all, there are no actually "comparable" analyses with which to compare OOCFA2, so these results are essentially offered in a vacuum. One issue is that—as stated in the discussion of this work's contributions (section 1.4)—this is the first analysis of its kind on Dalvik. Therefore no other approaches use a similar methodology (even as general as "abstract interpretation") and thus metrics or even just the intrepretation of them do not compare well between

approaches. Another issue is that—especially for performance data—it would simply be irresponsible to compare two approaches in, e.g., two different languages (never mind using different implementations) and expect the results to somehow convey information on the performance of one *algorithm* compared to another, versus simply the *implementations*.

The other caveat is regarding the benchmarks themselves. We have searched as much as we could, but we failed to uncover any "industry-standard" benchmarks which did not fall prey to at least one of the following problems:

- Most Java benchmarking suites feature a benchmarking framework and tooling infrastructure. This is nearly always using the reflective capabilities of the JVM in order to, e.g., dynamically load and invoke benchmarks. As we cannot (yet [subsubsection 6.1.2.1]) reason over JVM reflection—especially when it involves dynamically loading code—we cannot reason over the actual benchmarks themselves. If the dynamically loaded benchmark code is not otherwise patched (which, again, some benchmarks do), then we can instead manually add the code for analysis, given a local copy.

- Many Java benchmarks heavily depend on the input of some file. By "heavily depend", we mean that the intended nature of the benchmark and the metrics resulting from that nature are either *entirely* dependent upon the input or otherwise largely dependent. An example of this would be a benchmark whose configuration is set up through an XML file or a **Jython** [1] (Python running on the JVM) benchmark which interprets a Python file. A notable counterexample would be a typical FFT ("Fast Fourier Transform") benchmark. While its output obviously strongly depends upon its input, the nature of that output and the metrics which can be gleaned from the benchmark are largely independent of the actual data given; e.g., random input data would be about as informative as any other random input data with respect to the intent of the benchmark. Given such benchmarks—though we can safely and soundly abstract, e.g., the side-effecting operations of reading a file into unknown values—our abstract interpretation would not represent the "spirit" of the benchmark and its

algorithms terribly well.

- Most Java benchmarks are multi-threaded. Even if they can be single-threaded, they are written to cope with multi-threading. Thus, e.g., an integer taken from the command line is taken to be the number of threads with which to run. As we cannot abstractly determine this integer, we would end up with a situation where we either have to model an arbitrary number of threads or enforce single-threaded semantics on code expecting multi-threaded semantics. (For example the code could use work-queues for runnable subtasks; which we would end up understanding to be possibly arbitrary code running in possibly nondeterministic order.) Given that our algorithm does not attempt to reason over multiple threads (subsubsection 3.2.2.2), we would at best be misrepresenting the nature of the benchmark again.

3.4.2 Metrics

There are three primary metrics that we use to evaluate the accuracy of our algorithm. Though there are of course many ways we could attempt to measure this—simple numerical metrics or otherwise—we have chosen these metrics because they can convey an intuitive notion of "accuracy" very simply and directly vis-à-vis object-oriented code.

The first is the notion of **ilk accuracy**. This measures how accurate our algorithm is at resolving what the possible ilks of an abstract value are. This is important for a variety of reasons, but for the purposes of our evaluation (and arguably the most important reason in general) it particularly matters for virtual-method resolution. According to Java (and general OOP semantics), when a virtual method is called, the actual method which is called depends on the object it is being called upon; more specifically, its ilk. In this way, a subclass may "override" methods defined by its superclass—provided they are not **final**, in Java terms—assured that any of its instances will be directed to use the overridden version instead of the original one. Therefore upon attempting to abstractly resolve a virtual method call we take note of which and how many possible ilks an abstract value may take, using that as a metric.

The second is the derived notion of **virtual-method resolution accuracy**. Depending

on the ilks which were resolved from the abstract value in question, there may be multiple methods to which we could dispatch. Of course, several of these ilks may resolve the virtual method to the same actual method. However, even *if* the ilk can be narrowed to simply one possibility, the value itself may be unknown, and thus we cannot assume that the concrete value that this abstract value represents is actually of this *exact* ilk and not a subclass, which could potentially have overridden the method. Thankfully, this complication can be averted if it is known that the virtual method in question is final or that the ilk is final; there would then be no possibility of overriding. Thus we take note of the number of possible method resolutions we can make given a virtual method call and the ilks involved.

The final metric is that of **basic-block transition accuracy**. Depending on the result of a conditional test or a multi-way branch, there can be multiple possible successors to a basic block. Depending on how accurately the algorithm can track the possible concrete values an abstract value can take, it can more or less accurately determine which of those transitions is possible—which of those successors is reachable along that abstract path—and which are not. We take note of this information at each conditional and use that for this metric. Specifically, for conditionals which are binary (i.e., if-tests) we note whether one path can be entirely pruned or not. We could very well have noted how many of the possible concrete values involved could be valid along each path, but that would make for a much more complicated metric for what we feel to be little gain for the purposes of this experiment. The number of multi-way branches in these tests is so minimal that we exclude them from this discussions.

In terms of performance metrics, we simply note the average run time and the average memory usage of the implementation over five executions.

3.4.3 Tests

DaCapo consists of a set of open source, real world applications with non-trivial memory loads. It is widely considered the "gold-standard" of Java benchmarking, and thus we use it as one of our test programs.

MATSim provides a framework to implement large-scale agent-based transport

simulations. The framework consists of severel modules which can be combined or used stand-alone. To get a better understanding under which circumstances MATSim performs best, its creators created a simple benchmark that runs 20 iterations of a sample scenario with different settings. We include this benchmark as one of our test programs.

Each test program was analyzed in two runs: one **main-method-focused** and one **accessible-method-focused**. For main-method-focused tests, only the executable-accessible methods of the program are taken to be starting points for analysis—i.e., the standard Java "main method" having the signature `public static void/int main(String[])`. Accessible-method-focused tests instead consider all of the programs "accessible" methods, from an API point-of-view. This set includes all public methods and all protected methods of a non-final class. (Though technically using JVM reflection one can invoke even the private methods of a class, we ignore this case for obvious reasons, one of which is that it flies in the face of the conventional expectation of Java's semantics.)

Each of these tests was run with the implementation's full debugging enabled in order to allow verification of the results (resulting in, e.g., 657 MB of gzip-compressed debug output). This shouldn't have any real effect on the accuracy of the memory usage metrics, but it does slow the execution and so adversely affects the average run time. Scala version 2.10.0-RC5 was used on Oracle's JVM version 7.10, along with the following JVM options: `-ea -Xmx4g -Xss2m -XX:+UseThreadPriorities -XX:+UseCompressedOops -XX:+DoEscapeAnalysis`

3.4.4 Results

3.4.4.1 Ilk accuracy

Something which cannot be seen in the summarized data (Table 1a) is the nature of the possible ilks represented. In actuality, an interesting situation occurred: the only situations in which the algorithm resolved more than one ilk involved exceptionals. All of these exceptionals stemmed from exception-handling. This reflects the semantics of Java; whereas a method can only have one return type, it can throw any number of exception types, and not necessarily even through an explicit "throw". Thus, while the algorithm can precisely track the ilk of the return value of a method, it can easily conflate the possible ilks of

Table 1: OOCFA2 accuracy results

(a) Ilk accuracy: For each test, we show the number of virtual-method calls which were resolved to a certain number of ilks

	# Ilks															
Tests	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
DaCapo (main)	9150	19	131	794	117	—	88	—	1270	3491	—	—	—	—	—	—
DaCapo (accessible)	185795	56	234	808	162	8	250	8	1270	4693	9	—	—	—	—	—
MATSim (main)	1568	85	10	8	—	42	12	24	4	4	4	4	66	34	24	10
MATSim (accessible)	34335	126	84	86	637	464	200	2966	4	4	4	4	66	34	24	10

(b) Virtual-method resolution accuracy: For each test, we show the number of virtual-method calls which were resolved to single method

	# Virtual Methods
Tests	1
DaCapo (main)	15060
DaCapo (accessible)	193293
MATSim (main)	1899
MATSim (accessible)	39048

(c) Basic-block transition accuracy: For each test, we show the number of basic-block transitions which had two successors, then which ones were considered to have a certain number of reachable successors, then which ones resulted in a certain number of taken branches

	# Successors	# Reachable		# Branched		
Tests	2	1	2	0	1	2
DaCapo (main)	6380	93	6287	—	283	6097
DaCapo (accessible)	153018	81781	71237	315	82833	69871
MATSim (main)	125	7	118	—	8	117
MATSim (accessible)	35324	1341	33983	228	1635	33508

the exceptionals which may be uncaught by a method. (Exceptionals which are explicitly *thrown* by a method are tracked as precisely as normal return values.)

Even in the face of such conflation, the percentage of ilks that are resolved to exactly one is quite high: 61% for DaCapo (main), 96% for DaCapo (accessible), 83% for MATSim (main), and 88% for MATSim (accessible). Of course, in terms of non-exceptionals (i.e., the values that sincerely matter with regard to the benchmarks themselves), this percentage is 100% in all tests.

3.4.4.2 Virtual-method resolution accuracy

As curiously shown in the data (Table 1b), *all* virtual method invocations were precisely resolved for *all* tests. In reviewing the raw data, we believe that this is sincerely not in error. Let us review the situations for which a virtual method invocation can be precisely resolved:

1. The ilk is precisely resolved and the concrete value in question is known
2. The ilk is precisely resolved and the concrete value is *unknown*, but the virtual method in question is final, either itself or by proxy of its defining class being final.
3. The ilk has multiple possibilities, but they share the resolved implementation of the invoked virtual method (either because it is final or because none of the ilks overrides it).

Every single case of virtual-method resolution encountered in the tests fits one of those three conditions. It is likely that part of the fortune here is that these benchmarks—and benchmarks in general—tend not to stress virtual method resolution a great deal. Indeed, for performance reasons, they often attempt to avoid virtual methods which are difficult to statically determine.

3.4.4.3 Basic-block transition accuracy

The data in Table 1c shows that in most cases our algorithm was not able to *completely* eliminate one of the branches in a conditional. This is not entirely surprising, as it only

takes one unknown value (or—statistically—a large enough set of known values) to require both branches of the conditional to be explored due to possibility. Of note, the accessible-method-focused DaCapo test yielded a *very* large proportion of one-sided branches—larger than the remaining uncertain conditional branches. Upon examination of the debugging logs, it seems that this is because many of the accessible entry-points to DaCapo were shallow and involved object-equality comparisons. As they involve objects as arguments, these equality comparisons tend to involve far fewer possible concrete values and furthermore far fewer unknown values.

3.4.4.4 Performance

The most remarkable performance result (Table 2) is the amount of memory used during the accessible-method-focused tests, especially versus the main-method-focused tests. We can therefore conclude that the amount of memory used by the analysis is most affected by the number of starting points that it uses. This can be explained by remembering that—in order to soundly support (among other uses) the loop/recursion exploration scheme—the algorithm keeps track of the dependencies between abstract values. Thus essentially all abstract values remain strongly referenced during the analysis and so cannot be garbage-collected. Why this matters vis-à-vis a starting point is that a given chain of abstract values can share much ancestry given that its abstract interpretation started at the same point. Given a different starting point, a different "root" of the abstract value dependency graph is given and thus an entirely new space of memory is used. Even if an abstract value itself may still be canonically equivalent to another from another dependency graph (another starting point), due to our representation, they are two separate objects in the JVM. We believe this to be the largest factor in how much memory the analysis uses. We confirmed this suspicion through the examination of the JVM's profile of its heap space during the analysis's execution.

Perhaps with a programmatic notion of semantic aliasing (subsection 3.3.2), we could make it such that these dependency graphs were not entirely disjoint and an abstract value's identity could be tied to this notion, thus hopefully greatly reducing the number of actual

Table 2: OOCFA2 performance results

Test	Memory (MB)	Run time (M:S)
DaCapo (main)	353	0:36
DaCapo (accessible)	2490	3:52
MATSim (main)	577	2:00
MATSim (accessible)	3940	5:21

abstract values in-use.

Something else to notice in the performance data is that memory usage and run time do not necessarily correlate with the breadth of paths explored (as seen in Table 1c). Indeed, even though the MATSim tests featured far less breadth in paths explored than with DaCapo, they ran for *longer* and used *more* memory. Of course, this shouldn't be very surprising considering that all sorts of other factors could influence the performance far more than mere path-breadth: number of starting points (as discussed above); path-depth; complex manipulation of large, trackable state; etc. In this case, it is actually a factor of MATSim having more starting points and, moreover, doing much more array manipulation with trackable concrete values.

Chapter IV

SECURITY APPLICATION

This chapter describes the proof-of-concept Android security implementation. Bear in mind that it really is merely proof-of-concept, and thus restricted in scope and practical utility. In section 4.1 we describe the intuitions behind and the structuring of our permissions-analysis. Then in section 4.2 we provide an empirical evaluation of our permissions-analysis and discuss the results.

4.1 Permissions-checking scheme

Android's access control policies and permissions-using APIs are extremely under-documented. As such, it is nearly impossible for developers or researchers to surmise what permissions an application might require in order to function properly, short of actually testing the code in various scenarios. Worse, as previously discussed (subsubsection 1.3.1.2), the permissions and checking mechanisms are themselves ad-hoc, and so they cannot be automatically identified in the implementation code. This is also aside from the fact that many parts of the Android APIs are implemented in native code, exporting a Java shim, and thus the variety of unstructured means by which permissions could be checked is significantly larger, given the relative "freedom" of native code.

This all means that even crossing the technical hurdles of the permission-checking implementation—made significantly easier and more powerful through the use of OOCFA2—the largest hurdle remains in the form of statically identifying permissions-requiring API invocations at all. This is largely a case of needing the Android APIs to be suitably annotated with their permissions requirements. Even though this is a herculean task in and of itself, it is made more difficult given that some permission-requirements are non-trivial expressions. For example, the `java.lang.Runtime.exec` method accepts a `String` which is the system command to perform. An application requires the *DUMP* permission if attempting to invoke "`dumpsys`" or "`dumpstate`". Alternatively, it requires the *INSTALL_PACKAGES* permission if attempting to invoke "`pm install`". Thus, not only must all of the APIs be essentially manually annotated with the required permissions (something which should have best been done during development), an expressive and usable scheme of expressing possibly complex permissions must be developed and used.

4.1.1 Stowaway project

The **Stowaway** [9] [2] project is a static analysis tool and permissions map for identifying permission use in Android applications. The researchers behind the project needed to build a permissions map of the Android APIs, which we have used as a primary source in lieu of attempting to manually annotate the APIs ourselves. In order to build this permissions map, the researchers had to go through many hurdles to dynamically and in a relatively black-box manner discover which API uses required which permissions in which situations. [9, subsection 3.1]

4.1.2 Caveats

The Stowaway project's permissions map and our use of it does not come without caveats.

Of note, their permissions map only applies to Android 2.2, though most of the information is likely applicable to later versions. As verifying its later applicability in detail would require me essentially recreating their arduous testing process again, it would void the point of using their work and thus we have judged such concerns immaterial to the case study.

Furthermore, the format in which they present their data is relatively unstructured. In particular, they have not devised a structured means of expressing anything beyond simple, unconditional permissions-requirements. They include a "Note" column in the data, where they describe in prose any additional, conditional, or otherwise complex permissions-requirements for the API usage on that line. As this is presented in prose, we have elected to ignore any permissions listings which include a note, instead of attempting to encode some logic to reason over some of the provided prose. This is not too concerning as relatively few entries in the permissions map include notes (58 out of 1292). I have also taken the liberty of manually encoding some of the more complex permissions-usage conditionals directly into the analysis, as a proof of concept. (One such complex conditional is the previously used `java.lang.Runtime.exec` example.)

As previously discussed (subsubsection 3.2.2.6), OOCFA2 (and by proxy, this security analysis) does not currently encode any heuristics or knowledge of the Android environment.

In particular, neither analysis reasons over Intents [15]. The majority of the notes involve permissions checks which depend upon the Intent being used, reflecting the previously-noted dynamic and horizontal structuring and behavior of and between Android applications and the Android APIs.

Finally, also as previously noted (subsubsection 3.2.2.4), OOCFA2 does not currently reason over Java's run-time reflection mechanisms. As this is both a typical means of subverting static security analysis and not uncommon in Android-development practice, this lack of reasoning will currently create more false positives than necessary in the analysis many Android applications.

4.2 Empirical Evaluation

4.2.1 Tests

To test our application's ability to accurately detect permissions violations, we had it analyze several "malicious" applications we prepared. As OOCFA2 is implemented in terms of the dx tool (subsection 2.1.1), it currently requires JVM bytecode for the application one wishes to analyze. As the more traditional malware we would have wished to investigate is only available as already-translated DVM bytecode, this meant that we would be unable to analyze any of them. Thus we took a variety of sample applications from the Android SDK and made them malicious, subversive, or otherwise in misuse of permissions. We introduced such permissions violations as in our `java.lang.Runtime.exec` example.

The kinds of code we introduced for the permissions-violating method call were one or a combination of each of the following:

1. A plain, unadorned call in the normal flow of the code's execution
2. A call embedded within a conditional. These situations include:
   (a) An always-false conditional
   (b) A conditional whose branch depends on guaranteed-known values in the code
   (c) A conditional whose branch depends on unknown values in the code (e.g., inputs)

3. A call embedded in the static initialization of a class (whose initialization-time may be nondeterministic)

4. A permissions-violation which depends upon the value of one of the arguments to the method call. These situations include:

   (a) An always-known argument (e.g., a literal string)

   (b) An argument which depends upon guaranteed-known values in the code (e.g., a string composed of appending other literal strings)

   (c) An argument which depends upon unknown values in the code (e.g., a string input from the user)

4.2.2 Results

Following the same overall ordering as in our introduction of the kinds of code we used to test our application's permissions-violating capabilities, here we describe the results:

1. Plain, unadorned calls in the normal flow of the code's execution had no adverse affect on our ability to detect permissions violations.

2. Regarding calls embedded within a conditional:

   (a) Always-false conditionals were correctly never explored, and thus any permissions violations within the conditionals were ignored.

   (b) Conditionals depending upon guaranteed-known values yielded permissions-violations detection as one would expect:

      i. If all the known-value possibilities indicated the conditional would always be taken, our application concluded that there was a guaranteed permissions-violation.

      ii. If only some of the known-value possibilities indicated the conditional would be taken, our application concluded that there was a possible permissions-violation.

iii. If none of the known-value possibilities indicated the conditional would be taken, our application dutifully ignored any permissions-violations within the conditional's branch, as it did not explore the branch.

(c) Conditionals depending on unknown values would result in our analysis reporting that there was a possible permissions-violation.

3. A call embedded in the static initialization of a class resulted in permission-violations detection largely as one would expect:

   (a) If the class was never referred-to in the code, then its class-initialization code would never be executed and thus the permissions-violation would never occur.

   (b) If the class is referred-to at some point in the code, then its class-initialization code may be executed at various points in time, leading to a possible permissions-violation.

   (c) If the class is furthermore *guaranteed* to be referred-to in the code, then along all paths its class-initialization code will be executed. However, in this case our application reports a possible permissions-violation instead of a guaranteed one. It does not retroactively examine all the possible paths which lead to the permissions-violation; it instead merely notes the nondeterministic nature of the class initialization and judges the permissions-violation as merely "possible" should it occur. Keep in mind that OOCFA2 in general is not structured to keep track of all the intermediate states it has explored (unlike, e.g., CFA2), and thus there would be no reasonable way to achieve this required retrospective power.

4. Argument-dependent permissions-violations also resulted in detection as one would expect:

   (a) For always-known arguments, the application either reports that the permissions-violation is guaranteed or impossible, depending upon whether the arguments lead to a violation or not. If there are a mix of possible concrete values for these arguments, some which violate the permission and some which don't, then the

permissions-violation is merely possible.

(b) The same result comes of arguments which depend upon always-known values, assuming that OOCFA2 can reason over the function used on the values. An example where it can would be string appending; an example where it can't (currently) would be arbitrary-precision integer arithmetic.

(c) If an unknown argument is involved, the application must be conservative and report the permissions-violation as possible.

Combining all of these factors, our application would only ultimately report a "guaranteed" permissions-violation if all the proper factors were in place and the control path leading up to the violation did not involve any unknown values, i.e., it was absolutely certain that upon any "normal" execution of this application, there would be a permissions-violation. Likewise, our application would ultimately report a "possible" permissions-violation if this would be guaranteed if not for unknown values influencing the flow of control. It would report an "uncertain" permissions-violation if any of the factors considered the permissions-violation to be "possible" (it is impossible for any to report "impossible" unless all do). Finally, it would report an "impossible" permissions-violation if all of the factors considered the permissions-violation to be "impossible"—regardless of whether the control flow depends on unknowns.

Chapter V

RELATED WORK

5.1 Flow analysis

In this section we discuss the related work leading up to OOCFA2. We assume some degree of understanding of abstract-interpretation.

5.1.1 *k*-CFA

The ***k*-CFA** [22] family of algorithms were introduced by Shivers. They are phrased to operate upon higher-order functional programs, whereas OOCFA2 takes an object-oriented stance. These algorithms approximate the valid control-flow paths through the program as the set of all paths through a finite graph of abstract machine states, where each state represents a program point plus some amount of abstracted environment and control context.

However, the set of paths through a finite graph is a *regular language*, whereas the execution traces produced by recursive function calls are strings in a *context-free language*. Therefore the use of k-CFA on programs including recursive function calls permits execution paths which do not properly match calls with returns. This is particularly dangerous behavior in a higher-order functional setting, as functional values flowing down these spurious paths can lead to further "phantom" control-flow structures, along which functional values can then flow, and so on and so forth; this whole process then sharply decreases the precision of the analysis and likewise (by the logic used in section 1.2) leads to a sharp increase in the cost of the analysis.

5.1.2 CFA2

In order to remedy the problems of call/return matching problems of k-CFA, Vardoulakis and Shivers introduced **CFA2** [25]. This algorithm uses a *pushdown model* of abstract interpretation which can accurately resolve the call/return matching problems which arise out of k-CFA's regular-language-based modeling.

OOCFA2 is largely CFA2 as applied to an object-oriented environment, with some extensions (section 3.3) and, in terms of its primary implementation, tailored to Java's and the DVM's semantics. Notably unlike CFA2, our algorithm doesn't need to deal with

first-class control (or anything other than escape-continuations), and thus we don't use this machinery from CFA2 [26].

5.2 Android security

In this section we discuss some of the state-of-the-art alternatives and solutions to Android security, as proposed in the literature and implemented.

ScanDal [16] is a sound and automatic static analyzer for detecting privacy leaks in Android applications. Like OOCFA2, it too is an abstract interpretation. However, it works over the DOP instead of the ROP and is a first-order analysis instead of a higher-order one. It is an information-flow analysis which determines if there exists any flow of data from a sensitive source through a sensitive sink. As discussed in subsubsection 6.1.3.1, we have the ability to do the same and plan to do so in future work. Furthermore, their analysis is much more tailored to Android, with Android-specific abstract-interpretation including callbacks and event-handling. This, too, is something with which we hope to extend our analysis (subsubsection 6.1.2.2). Nevertheless, our analysis is fundamentally more precise and powerful, thus we expect it to perform much better than ScanDal once it is specifically tailored to Android.

TaintDroid [7] monitors Android applications at run-time, sacrificing run-time performance for the ability to track how applications leak private information. Thus, it is a dynamic solution, in contrast to our static solution. We feel that its 14% CPU overhead is inappropriate for an embedded environment, and that a purely static approach is a better fit. Of course, it is also possible for the approaches to complement each other; if OOCFA2-based security analyses indicate that the application is safe, then it can be run without the overhead of TaintDroid—otherwise we could use TaintDroid to still allow the statically-unsafe execution of the application.

Enck et al. [8] used decompiling techniques, some automated tools, and manual inspections to determine the safety of Android applications. They studied 1100 popular free Android applications, showing that they often misuse users' private data. As for their decompilation, they designed and implemented a DOP-to-Java-source decompiler,

then analyzing the Java source. However, they failed to recover about 5% of the classes in the applications they analyzed. As our solution does not depend on decompilation, we avoid such problems entirely.

Chapter VI

CONCLUSION & PERSPECTIVE

In this work we've introduced the OOCFA2 algorithm, an implementation of it, and a proof-of-concept security-oriented application of it. We've also introduced a formal semantics for Dalvik's ROP.

This is the first PDA-based higher-order flow analysis in an object-oriented setting. With respect to the modeling of the Dalvik virtual machine, this is the first application of a higher-order flow analysis to an assembly language. We discussed the algorithm in chapter 3 and in particular our novel extensions to flow-analysis in section 3.3.

This work also features the first (published) formal model of the Dalvik virtual machine and its semantics. As far as we are aware, it is also most definitely the first formal model targeting Dalvik's ROP (cf. section 2.1 and section 2.2). We discussed this model—named Welterweight Dalvik—in section 2.3.

The Android-security-focused proof-of-concept application also is the first (published example) of its kind. We discussed this application in chapter 4. The use of a such a powerful higher-order flow analysis allows one to apply its knowledge to create a wide variety of powerful and practical security-analysis "front-ends"—not only the permissions-checking analysis in this work, but also, e.g., information-flow analyses (subsubsection 6.1.3.1).

We empirically evaluate OOCFA2's accuracy and performance (section 3.4) to prove its practical viability. We also evaluated the proof-of-concept security analysis' accuracy as directly related to OOCFA2 (section 4.2); this shows promising results for the potential of building security-oriented "front-ends" atop OOCFA2.

6.1 Future work

6.1.1 Flow analysis

6.1.1.1 Loop-indexed semantic aliasing

As discussed in subsection 3.3.2, semantic aliasing is an invaluable concept for any consumer of the output of OOCFA2 to make use of its state-space representations. This utility can be further extended by, e.g., discovering structured loops and identifying a set of otherwise-undesirable semantic aliases as related across an iteration of such a loop. The truly interesting point comes when considering *nested* loops; each level of nesting would

equate to an "axis" in the iteration-space, with each alias noted at its respective point in this iteration-space. At first pass, this would allow a user of the analysis to see the "progression" of an aliased value along one of the axes as the difference between the value along the iteration. They could also selectively "compress" an axis (or multiple) in the space for some aliased valued, forcing all such aliases to be considered one semantic alias representing all of the possible values the alias could be with respect to a loop. We are uncertain of the particular uses of the ability to view the data in this manner, but it certainly seems to be potentially quite useful.

6.1.2 Android ecosystem

6.1.2.1 Reflection-reasoning

Since we have an essentially whole-system view of the APIs and application, we can abstractly interpret the string-based reflection methods as reflecting upon our own "database" of classes, methods, and fields.

The analysis currently supports and encourages the loading of external java archives, which are injected into the analysis' JVM and introspected-upon. This is used to, e.g., load the Android standard libraries and introspect upon them, allowing for the more precise identification of classes, methods, and fields used in the analyzed program. Should a reflective method be encountered, this large space of introspection—along with the introspection built from the analyzed code itself—could be consulted to satisfy (or at least refine the possible outcomes of) the reflected method.

6.1.2.2 Android-specific heuristics

As previously discussed (in subsubsection 3.2.2.6), the analysis does not currently encode any heuristics or knowledge of the Android environment. Some particular highlights of Android-ignorance include:

- Android-specific system, development, and application framework concepts [12], such as:
  - Activities

  - Intents
  - Content Providers
  - bound services
  - component processes
- Android-specific metadata [12], such as AIDL and the "AndroidManifest.xml" file
- Application resources [13]
- Common Android development practices

In particular, the lack of reasoning over the Intents system and its Intents, Activities, Content Providers, and bound services is the most pressing issue in this regard. The system is relatively well-documented with plentiful metadata describing each component's behavior and requirements. There are also Android system-provided services with guaranteed behavior and standard Intents which would be very valuable to reason over.

6.1.3 Further security uses

6.1.3.1 Information-flow analyses

An **information-flow** analysis involves analyzing the data dependencies in a program and asserting that no information-flow violations occur, wherein a producer of sensitive information (a **source**) somehow provides data to an illegitimate consumer of said sensitive information (a **sink**).

As discussed in subsection 3.3.1, OOCFA2 currently tracks the dependencies between values during an abstract interpretation, in order to properly implement the cleanup phase of the loop-phasing technique. This corresponds cleanly to the interests of an information flow analysis; OOCFA2 even tracks the *control* dependencies between values, not just the *data* dependencies. The machinery which exists is so perfect for this use that one should be able to write a simple information-flow security analysis front-end to OOCFA2, which is simply aware of what the sources, sinks, and types of information of interest are in an Android setting (or alternatively be able to dynamically identify sources and sinks). This is left for future work.

Bibliography

[1] "Jython: Python for the Java Platform," 2012. [Online; accessed 30-November-2012] `http://jython.org/`. 61

[2] "Stowaway: A static analysis tool and permission map for identifying permission use in android applications," 2012. [Online; accessed 24-October-2012] `http://www.android-permissions.org/`. 71

[3] Allen, E., Hilburn, J., Kilpatrick, S., Luchangco, V., Ryu, S., Chase, D., and Steele, G., "Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, (New York, NY, USA), pp. 973–992, ACM, 2011. 42

[4] Android Open Source Project, "Bytecode for the Dalvik VM," 2007. [Online; accessed 13-October-2012] `http://source.android.com/tech/dalvik/dalvik-bytecode.html`. 11

[5] Android Open Source Project, ".dex — Dalvik Executable Format," 2007. [Online; accessed 13-October-2012] `http://source.android.com/tech/dalvik/dex-format.html`. 11

[6] Apache Software Foundation, "Apache Harmony," 2012. [Online; accessed 9-October-2012] `https://harmony.apache.org`. 4

[7] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N., "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010. 78

[8] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S., "A study of android application security," in *Proceedings of the 20th USENIX Security Symposium*, SEC'11, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2011. 78

[9] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D., "Android Permissions Demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, (New York, NY, USA), pp. 627–638, ACM, 2011. 7, 55, 71

[10] Findler, R. B. and Klein, C., "Redex: Practical Semantics Engineering," 2013. [Online; accessed 14-January-2013] `http://docs.racket-lang.org/redex/index.html`. 17

[11] Google, "Android," 2012. [Online; accessed 12-October-2012] `https://developer.android.com/index.html`. ix, 2, 4, 54

[12] Google, "App Components," 2012. [Online; accessed 12-October-2012] `https://developer.android.com/guide/components/index.html`. 82, 83

[13] Google, "App Resources," 2012. [Online; accessed 12-October-2012] https://developer.android.com/topics/resources/index.html. 83

[14] Google, "Dalvik Technical Information," 2012. [Online; accessed 12-October-2012] http://source.android.com/tech/dalvik/index.html. 4

[15] Google, "Intents and Intent Filters," 2012. [Online; accessed 12-October-2012] https://developer.android.com/guide/components/intents-filters.html. 72

[16] Kim, J., Yoon, Y., Yi, K., and Shin, J., "ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications," *Mobile Security Technologies (MoST)*, 2012. 78

[17] Linux Foundation, "The Linux Kernel Archives," 2012. [Online; accessed 9-October-2012] https://www.kernel.org. 3, 4

[18] Might, M., *Environment analysis of higher-order languages*. PhD thesis, Atlanta, GA, USA, 2007. AAI3271560. 3

[19] Oracle, "Java Language and Virtual Machine Specifications," 2012. [Online; accessed 14-October-2012] http://docs.oracle.com/javase/specs/. 11

[20] Oracle, "Java Overview," 2012. [Online; accessed 10-October-2012] http://www.oracle.com/us/technologies/java/overview/index.html. 4

[21] Oracle, "Unchecked Exceptions—The Controversy," 2013. [Online; accessed 14-January-2013] http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html. 54

[22] Shivers, O., *Control-Flow Analysis of Higher-Order Languages (or Taming Lambda)*. PhD thesis, May 1991. 77

[23] Smieh, "File:Android-System-Architecture.svg," 2012. [Online; version at 06:38, 29-June-2012] https://en.wikipedia.org/wiki/File:Android-System-Architecture.svg. 5

[24] Vardoulakis, D., "Polyvariant Flow Analysis for JavaScript with CFA2." 2011. iv, 60

[25] Vardoulakis, D. and Shivers, O., "CFA2: a Context-Free Approach to Control-Flow Analysis," Tech. Rep. NU-CCIS-10-01, Northeastern University, January 2010. iv, 60, 77

[26] Vardoulakis, D. and Shivers, O., "Pushdown flow analysis of first-class control," *SIGPLAN Not.*, vol. 46, pp. 69–80, September 2011. 78

[27] Wikipedia, "Android (operating system): Market share and rate of adoption — Wikipedia, the free encyclopedia," 2012. [Online; accessed 12-October-2012] https://en.wikipedia.org/wiki/Android_operating_system#Market_share_and_rate_of_adoption. 3

[28] École Polytechnique Fédérale de Lausanne (EPFL), "Scala," 2013. [Online; accessed 14-January-2013] http://www.scala-lang.org. 41